UNIVERSITÉ DE LORRAINE

Draft v1.4 – October 30, 2026

Université de Lorraine

École doctorale IAEM, LORIA

Formal Methods and Real-World Cryptographic Protocols: Advancing the Verification and Testing Frontiers for Specifications and Implementations

Par Lucca HIRSCHI

Document de synthèse
présenté pour obtenir
L'Habilitation à Diriger des Recherches
mention informatique

Présentée et soutenue publiquement le 2 novembre 2026

Devant un jury composé de :

• Gilles BARTHE, Directeur scientifique *MPI-SP, Allemagne et IMDEA, Espagne*	Rapporteur
• Karthik BHARGAVAN, Chief Research Scientist *Cryspen, France*	Rapporteur
• Ralf KÜSTERS, Professeur *Université de Stuttgart, Allemagne*	Rapporteur
• David BASIN, Professeur *ETH Zurich, Suisse*	Examinateur
• Anne CANTEAUT, Directrice de recherche *Inria Paris, France*	Examinatrice
• Jean-Yves MARION, Professeur *Université de Lorraine, LORIA, France*	Examinateur
• Catuscia PALAMIDESSI, Directrice de recherche *Inria Saclay, LIX, France*	Examinatrice
• Olivier PEREIRA, Professeur *UCLouvain, Belgique*	Examinateur
• Thorsten HOLZ, Directeur scientifique *MPI-SP, Allemagne*	Membre invité
• Steve KREMER, Directeur de recherche *Inria Nancy - Grand-Est, LORIA, France*	Membre invité

Summary / Résumé

Titre : Protocoles cryptographiques et méthodes formelles : repousser les limites de la vérification des spécifications et du test des implémentations

Mots clefs : Protocoles cryptographiques ; Analyse de sécurité ; Méthodes formelles ; Vérification symbolique ; Modèle de Dolev-Yao ; Test logiciel ; Fuzzing

Résumé : Les protocoles cryptographiques sont les garants de nos échanges numériques sécurisés, piliers technologiques de notre société de l'information. L'histoire répétée d'attaques exploitant les erreurs de conception et d'implémentation de ces protocoles n'a de cesse de démontrer leur fragilité. Ce manuscrit aborde un défi central de la sécurité informatique : comment obtenir de meilleures garanties pour les protocoles cryptographiques ?

Premièrement, nous présentons des avancées fondamentales en méthodes formelles, introduisant de nouvelles techniques de modélisation, des algorithmes de vérification et des méthodologies de preuve dans le modèle Dolev-Yao visant à garantir formellement la sécurité des protocoles cryptographiques face à leur modèle d'attaquant. Deuxièmement, nous les mettons en application à grande échelle dans des analyses de sécurité de protocoles largement déployés. Nous détaillons notamment la découverte de vulnérabilités critiques et leur correction dans des protocoles majeurs tels que les standards industriels de la téléphonie mobile (4G et 5G) et des systèmes de contrôle industriel (OPC UA), ainsi que le système de vote électronique français (FLEP) déployé pour les élections législatives. Enfin, nous ciblons l'écart entre la vérification formelle des spécifications de protocoles et la sécurité de leurs implémentations en introduisant une nouvelle approche de test par fuzzing guidée par le modèle formel de Dolev-Yao. Ce nouveau paradigme de test logiciel permet la découverte automatique d'attaques logiques causées par des bugs dans des implémentations de protocoles cryptographiques complexes. Nous démontrons son efficacité, notamment par la détection de plusieurs nouvelles vulnérabilités et de dizaines de bugs dans des implémentations de protocoles parmi les plus testés (TLS et SSH).

Title: Formal Methods and Real-World Cryptographic Protocols: Advancing the Verification and Testing Frontiers for Specifications and Implementations

Keywords: Cryptographic protocols; Security analysis; Formal methods; Symbolic verification; Dolev-Yao model; Software testing; Fuzzing

Abstract: Cryptographic protocols are the backbone of secure digital communication and a key technological pillar of our information society. Yet a long history of attacks exploiting both design flaws and implementation bugs has repeatedly exposed their fragility. This manuscript addresses a central challenge in computer security: how can we achieve higher assurance for cryptographic protocols?

First, we present foundational advances in formal methods, introducing new modeling techniques, verification algorithms, and proof methodologies within the Dolev-Yao model that aim to provide formal security guarantees for cryptographic protocols against a powerful network attacker. Second, we put these methods into practice through large-scale security analyses of widely deployed protocols. In particular, we detail the discovery and remediation of critical vulnerabilities—since fixed—in major protocols, including mobile telephony standards (4G and 5G), industrial control system standards (OPC UA), and the French electronic voting system (FLEP) deployed for national elections. Finally, we address the gap between the formal verification of protocol specifications and the security of their real-world implementations by introducing a novel Dolev-Yao model-guided fuzzing approach. This new software testing paradigm enables the automatic discovery of logical attacks caused by implementation bugs in complex cryptographic protocols. We demonstrate its effectiveness through the detection of several previously unknown vulnerabilities and dozens of bugs in implementations of TLS and SSH—some of the most extensively tested protocols.

Contents

Summary / Résumé i

Contents iii

List of Figures viii

List of Tables ix

List of Acronyms x

1. Introduction 2
- 1.1. When Cryptographic Protocols Fail . . . 4
- 1.2. State of the Art in Cryptographic Protocol Analysis . . . 6
  - 1.2.1. Design-level Verification . . . 6
  - 1.2.2. Implementation-level Verification . . . 8
  - 1.2.3. Testing and Fuzzing . . . 9
- 1.3. Research Journey Overview . . . 10
  - 1.3.1. Part A: Advancing Formal Methods for Cryptographic Protocol Security Verification 11
  - 1.3.2. Part B: Large-Scale Real-World Protocol Analyses . . . 12
  - 1.3.3. Part C: Formal Models Meet Fuzz Testing . . . 13

2. Background 18
- 2.1. Standard Cryptographic Primitives . . . 18
- 2.2. Security Protocols . . . 19
  - 2.2.1. Sigma', an Authenticated Key Exchange Protocol . . . 20
  - 2.2.2. The Noise Family of AKE Protocols . . . 20
- 2.3. Dolev-Yao Model . . . 24
  - 2.3.1. Cryptographic Messages Models . . . 24
  - 2.3.2. Protocol Models . . . 25
  - 2.3.3. Security Property Models . . . 27
- 2.4. Automated Verification . . . 28
  - 2.4.1. The Tamarin Prover . . . 29
  - 2.4.2. The ProVerif Prover . . . 33
  - 2.4.3. Other Approaches and Tools . . . 35

A. Advancing Formal Methods for Cryptographic Protocol Security Verification 36

3. New Modeling Techniques 41
- 3.1. Hash Functions Weaknesses in the Real World . . . 42
- 3.2. Threat Models for Weak Hash Functions . . . 43
  - 3.2.1. Example: Hash Transcript Collisions and Chosen-Prefix Collision (CPC) . . . 43
  - 3.2.2. Lattice of Adversarial Capabilities . . . 44
- 3.3. Automated Analysis with Weak Hash Functions . . . 46
- 3.4. Analyses Results . . . 48
- 3.5. Other Modeling Technique Advances . . . 49
  - 3.5.1. ID-Based Cryptographic Protocols . . . 49
  - 3.5.2. Modeling Distributed Cryptographic Protocols . . . 50

4. Improving Verification Tools 52
- 4.1. Equational Theories . . . 52
  - 4.1.1. Modeling Exclusive-Or in Tamarin . . . 52

4.1.2. Modeling Purely Associative Symbols in ProVerif and Tamarin . . . 54
4.2. Equivalence Verification . . . 55
4.2.1. Partial Order Reduction Techniques for the Bounded Number of Sessions Setting . . 55
4.2.2. Provable Sufficient Conditions for the Unbounded Number of Sessions Setting . . . 56

5. New Proof Methodologies **58**
5.1. Dealing with a Large Collection of Verification Goals . . . 59
5.1.1. Noise Framework: Systematically Compare Protocols under Various Threat Models 59
5.1.2. Hash Weaknesses: Measuring Resistance to Various Threats and Weaknesses 60
5.1.3. OPC UA: Taming Non-Termination With Protocol Feature-Specific Proof Heuristics 60
5.2. Defining a Lattice of Verification Goals . . . 61
5.3. Static Analysis . . . 64
5.4. Lattice Exploration . . . 65
5.5. Example of Results . . . 67
5.6. Proof Helper Methodology for ProVerif . . . 68

B. Large-Scale Protocol Analyses **71**

6. Repairing an International ICS Standard **75**
6.1. Open Platform Communication Unified Architecture (OPC UA) Protocol . . . 77
6.1.1. Overview . . . 77
6.1.2. Protocol Description . . . 78
6.1.3. Security Goals . . . 81
6.2. OPC UA Formal Model . . . 83
6.2.1. Modeling in the Applied Pi Calculus . . . 83
6.2.2. Formal Security Properties . . . 84
6.3. Analysis Results . . . 84
6.3.1. Client Impersonation in ECC . . . 85
6.3.2. KCI: User Impersonation . . . 88
6.3.3. Session Hijack by Reopening or Switching . . . 90
6.3.4. KCI: Session and User Confusion . . . 91
6.3.5. Other Findings . . . 92
6.4. Lessons Learned and Discussions with the OPC Foundation . . . 94
6.4.1. Key Lessons on Protocol Design . . . 94
6.4.2. Fixes of the Main Remaining Attacks in v1.05.07 . . . 94
6.4.3. Responsible Disclosure and Actions Taken by the OPC Foundation . . . 95
6.5. Conclusion . . . 97

7. Challenging and Improving the 4G and 5G Mobile Communication Standards **99**
7.1. Simplified Overview of 5G Authentication and Key Agreement (AKA) . . . 100
7.1.1. Architecture . . . 101
7.1.2. Authentication Protocols . . . 101
7.2. Formal Analysis of 5G AKA . . . 104
7.2.1. Summary of our 5G AKA Analysis . . . 105
7.2.2. Selection of Key Results of our 5G AKA Analysis . . . 106
7.3. New Privacy Risks and Practical Attacks . . . 109
7.3.1. Privacy Risks over WiFi . . . 109
7.3.2. Privacy Risks of AKA in 3G, 4G, and 5G . . . 109
7.4. Lessons Learned and Discussions with GSM Association (GSMA) and 3rd Generation Partnership Project (3GPP) . . . 115
7.4.1. Recommendations for the 5G AKA Standard . . . 115
7.4.2. Recommendations Regarding Privacy . . . 116
7.4.3. Challenges with Formal Analysis of Privacy Properties for AKA Protocols . . . 117
7.4.4. Responsible Disclosure and Further Impact . . . 118

8. Breaking and Fixing E-voting and French National Election **122**
8.1. Introduction . . . 122

8.2. Context . . . 124
8.2.1. Architecture . . . 124
8.2.2. Security Goals and Threat Model . . . 125
8.3. Reverse the Protocol . . . 127
8.4. Vulnerabilities, Attacks, and Fixes . . . 130
8.4.1. Vulnerabilities . . . 130
8.4.2. Attacking and Fixing Verifiability . . . 132
8.4.3. Attacking and Fixing Ballot Privacy . . . 134
8.5. Recommendations and Lessons Learned . . . 139
8.6. Breaking and Fixing Other E-Voting Components . . . 141
8.7. Responsible Disclosure and Further Impact . . . 142
8.8. Conclusion . . . 145

C. Formal Models Meet Fuzz Testing 146

9. Dolev-Yao Fuzzing 154
9.1. Background . . . 154
9.1.1. Case Study: The Transport Layer Security (TLS) Protocol . . . 154
9.1.2. Fuzzing . . . 155
9.1.3. Related Work . . . 156
9.2. Dolev-Yao Model Abstraction Amenable to Fuzzing . . . 157
9.2.1. Term Algebra . . . 157
9.2.2. DY Traces . . . 158
9.2.3. DY Security Properties . . . 161
9.3. DY Fuzzing Design . . . 161
9.3.1. Mapper . . . 162
9.3.2. Harness . . . 164
9.3.3. DY Traces and Seed Corpus . . . 164
9.3.4. Mutations . . . 165
9.3.5. Security Policies and Objective Oracle . . . 167
9.3.6. Big Picture: The DY Fuzzing Loop . . . 167
9.4. Brief Overview of Differential DY Fuzzing . . . 167
9.4.1. Architecture and Differential Oracle . . . 168
9.4.2. Key Challenges and Solutions . . . 169

10. Implementation, Evaluation, and Case Studies 171
10.1. The PUFFIN DYF Library and its Fuzzers . . . 171
10.1.1. Modular Architecture . . . 172
10.1.2. Implementation Challenges and Features . . . 173
10.1.3. Implementing Differential DY Fuzzing in PUFFIN . . . 174
10.2. Evaluation of DY Fuzzing . . . 174
10.2.1. Finding DY Attacks . . . 174
10.2.2. Coverage Comparison . . . 178
10.3. Evaluation of Differential DY Fuzzing . . . 180
10.4. Summary of Findings and Responsible Disclosure . . . 183

11. General Perspectives 185
11.1. Formal Verification Perspectives . . . 187
11.1.1. Unified Lattice Framework . . . 187
11.1.2. Composite Arguments and Lattices . . . 188
11.1.3. Optimizing Verification Using Earlier Results . . . 189
11.2. E-Voting Protocol Perspectives . . . 190
11.3. Dolev-Yao (DY) Fuzzing Perspectives . . . 191
11.3.1. Improve the DY Fuzzer Feedback . . . 191
11.3.2. Broaden the Scope of the DY *Objective Oracle* . . . 192
11.3.3. Semi-automating *Mapper* Construction for New Protocols . . . 193

11.3.4. Combining DY and Bit-Level Fuzzing 194
11.3.5. Advancing Differential DY Fuzzing 195
11.3.6. Engineering Infrastructure and Longer-Term Maintenance 195
11.4. Bridging the Gap Between DY Fuzzing and Verification 196
11.4.1. Hybrid Framework and Fuzzing 196
11.4.2. Model Extraction and Refinement 197

List of Figures

1.1. *Untitled* — Calder, 1942 . . . 3
1.2. Taxonomy of vulnerabilities in cryptographic protocol specifications and implementations (TLS examples). . . . 5
1.3. Intuitively visualizing the trade-off between computational and symbolic models . . . 7
1.4. Research Journey Overview . . . 14

2.1. The Sigma' protocol [BL16] . . . 20
2.2. Three Noise handshakes . . . 21
2.3. Alice & Bob notation for Noise handshakes . . . 22
2.4. The Tamarin prover logo. . . . 29
2.5. The ProVerif prover logo. . . . 29
2.6. Tamarin's verification workflow . . . 29
2.7. Example execution of $P_{Sigma'} \cup MD$ . . . 30
2.8. Message deduction graphs for pairing . . . 31
2.9. Normal deduction rules ND . . . 32
2.10. ProVerif's verification workflow . . . 33
2.11. Lattice of security guarantees for Noise protocols . . . 39

3.1. CPC attack on Sigma' . . . 44
3.2. Lattice of adversarial capabilities . . . 45
3.3. New CPC attack on Sigma' . . . 49

4.1. Example 4.1.1: Infinite chain of deconstruction rules. . . . 53

5.1. Atomic adversary capabilities; the set thereof is denoted A. . . . 62

6.1. Overview of the OPC UA protocol . . . 78
6.2. Overview of the OPC UA state machine . . . 78
6.3. OPC UA secure channel opening and communications . . . 80
6.4. OPC UA session creation, activation, and user requests . . . 80
6.5. Query $Conf_{\mathrm{C}}$: confidentiality of user requests . . . 84
6.6. Query $Conf_{\mathrm{Pwd}}$ modeling confidentiality of passwords. . . . 84
6.7. Query Agr_{S}: data and entity authentication . . . 84
6.8. Client impersonation attack (ECC/NoAA) . . . 88
6.9. KCI user impersonation attack . . . 89
6.10. Session hijacking attack (channel reopen) . . . 90
6.11. Session confusion attack . . . 91

7.1. Mobile communications architecture . . . 101
7.2. 5G AKA authentication initiation . . . 102
7.3. 5G AKA protocol (simplified) . . . 103
7.4. Sequence Number Inference Attack . . . 111
7.5. Experimental setup for SQN inference . . . 114

8.1. Organization of the French legislative elections . . . 125
8.2. Description of the 2022 French Legislative Election Protocol . . . 128
8.3. puffin fuzzer: Finding all your fishy code . . . 149

9.1. Generic fuzzer architecture . . . 155
9.2. Overview of the DY fuzzing loop . . . 163
9.3. Differential Objective Oracle workflow . . . 168
9.4. Differential Harness Architecture . . . 168

10.1. ProtocolBehavior trait (simplified) . . . 172

10.2. AGENT_INTERFACE C ABI (simplified) 172

11.1. Mysteries of the future. The “Dawn of Man” sequence from *2001: A Space Odyssey* (1968). . . . 185

List of Tables

2.1. Noise options and nomenclature 21

3.1. Weaknesses of widely used hash functions 43
3.2. Collision-relations $\sim_c$ by adversarial capability 46
3.3. Sigma' security analysis. 48

5.1. Strongest threat models (STM) for secrecy 67
5.2. Impact of attacks on a national e-voting system 73

6.1. Security analysis summary for OPC UA 85
6.2. Summary of the attacks found that affect OPC UA v1.05.03. 87

7.1. Minimal assumptions for 5G AKA authentication 107
7.2. Minimal assumptions for 5G AKA secrecy 107
7.3. Minimal assumptions for 5G AKA (UE–SN auth.) 108

8.1. Main cryptographic values involved in the French Legislative E-Voting Protocol (FLEP). 127
8.2. Summary of attacks and fixes in FLEP 131

9.1. Summary of action-level and term-level mutations for DY fuzzing. 165

10.1. Lines of code: PUFFIN and its fuzzers 171
10.2. Vulnerabilities (re)discovered with TLSPUFFIN 176
10.3. CVEs found with PUFFIN in Differential Dolev-Yao Fuzzing (DDYF) mode 181

List of Acronyms

3GPP 3rd Generation Partnership Project. iv, 13, 76, 103, 104, 114, 119, 121, 123–125

AC Associative-Commutative. 26, 31
AEAD Authenticated Encryption with Associated Data. 20, 22–24
AKA Authentication and Key Agreement. iv, 3, 11–13, 16, 51, 76, 78, 103–124
AKE Authenticated Key Exchange. 21
ANSSI Agence nationale de la sécurité des systèmes d'information. 128, 132, 135, 147–149
ASAN AddressSanitizer. 157, 170, 174, 183

CDH Computational Diffie-Hellman. 20
CNIL Commission nationale de l'informatique et des libertés. 130, 131, 138, 145
CPC Chosen-Prefix Collision. iii, vii, 43, 45–47, 49–51
CVE Common Vulnerabilities and Exposures. 15, 80

DDYF Differential Dolev-Yao Fuzzing. ix, 14, 15, 158, 159, 166, 174–176, 179, 182, 184, 188–192, 194, 201, 203, 204
DH Diffie-Hellman. 83–86, 155
DNF Disjunctive Normal Form. 66
DOS Denial-of-service attack. 90
DSL Domain Specific Language. 172, 176, 180, 182
DY Dolev-Yao. v, 8, 11, 14–17, 21, 25, 28, 30, 154–156, 164, 193, 194, 199–205
DYF Dolev-Yao model-guided Fuzzing. 14, 15, 156, 158, 159, 161, 174, 179, 182, 188, 189, 203, 204

ECC Elliptic Curve Cryptography. 79–81, 90
ECDH Elliptic Curve Diffie–Hellman. 81
EFA Ministry Europe and Foreign Affairs French Ministry. 128, 130, 132, 135, 145–149
EMV Europay Mastercard Visa. 3, 5, 6

FLEP French Legislative E-Voting Protocol. ix, 11, 13, 14, 16, 77, 127–136, 138, 142, 144–148, 150, 155
FVP Finite Variant Property. 26, 31, 32, 36

GCVE Global CVE Allocation System. 13, 80, 89, 99, 101
GSMA GSM Association. iv, 76, 103, 104, 113, 114, 119, 121, 123–125

HMAC Keyed-Hash Message Authentication Code. 82–84
HSM Hardware Security Module. 97

ICS Industrial Control Systems. 13, 16, 77–79, 204
IETF Internet Engineering Task Force. 21
IPC Identical-Prefix Collision. 46, 49
IPsec Internet Protocol Security. 79
IT Information Technology. 79

JWT JSON Web Token. 82

KCI Key Compromise Impersonation. 24, 25, 61, 64, 66, 92, 95
KDF Key Derivation Function. 22, 23, 83, 106

LLM Large Language Model. 180, 193–195, 203

MAC Message Authentication Code. 20, 21, 106
MD Merkle-Damgård. 43, 47, 49, 57
MIM Machine-in-the-Middle. 5, 11, 45, 46, 51, 85, 131, 155, 161, 165, 166, 172, 184, 185, 200
MLS Messaging Layer Security. 9
MSR Multiset Rewriting Rule. 27–29, 31, 32, 34

OPC UA Open Platform Communication Unified Architecture. iv, vii, ix, 3, 5, 11, 13, 16, 72, 76–83, 85–89, 91–93, 96–102, 157, 179, 198, 202, 204
OT Operational Technology. 79, 82

PCS Post-Compromise Security. 61
PFS Perfect Forward Secrecy. 61, 66, 79, 81, 86, 88, 110, 111
PKI Public Key Infrastructure. 21, 22, 81, 87
PoC Proof-of-Concept. 117, 157, 158, 179
PPTM Probabilistic Polynomial-time Turing Machine. 7, 8
PQ Post-Quantum. 8
PSK Pre-Shared Key. 22, 23, 70, 71, 161, 165, 170, 184
PUT Program Under Test. 10, 14, 156–159, 162–164, 166–172, 174–176, 179–183, 185, 186, 188–191, 200, 202–205

QUIC Quick UDP Internet Connections. 8, 9

RCE Remote Code Execution. 6, 184
ROM Random Oracle Model. 43, 44, 46–48

SCADA Supervisory Control and Data Acquisition. 79, 81
STM Strongest Threat Model. 66, 67, 70, 71

TLS Transport Layer Security. v, 3, 5–10, 14, 15, 79, 81, 155, 157–159, 161, 162, 165, 166, 168, 169, 171–174, 179, 182, 184, 187, 199, 204, 205

UASC UA Secure Conversation. 79, 81, 83
UC Universal Composability. 122
UKS Uknown Key-Share. 61, 97

XOR Exclusive OR. 12, 195

ZKP Zero-Knowledge Proof. 134–137, 143

1. Introduction

Our modern society has largely migrated to the digital realm, in what is often described as an *information society*, undergoing a massive and rapid digitalization of our social, professional, economic, and civic interactions, albeit unevenly across regions and social groups.[1] In a single day, it is not uncommon to message friends and family through a secure messaging app, unlock one's home with a smart door lock, pay for coffee by tapping a card, work remotely across multiple online services, and manage life savings through a banking application. This digital transformation rests on a fundamental assumption: that our interactions are secure. We trust that no one else is reading our messages, that the bank website is genuine, and that our smart door lock only opens for us.

1: This transformation is at times imposed —*e.g.,* through mandatory digital channels for certain services— and is not uniformly benign: it has also facilitated documented forms of mass surveillance, algorithmic manipulation, and growing market concentration among a handful of large digital platforms [Zub19].

This trust is not magic; it is rooted in the invisible algorithms that govern our digital interactions, known as *cryptographic protocols*[2]. These are distributed algorithms that run whenever digital devices wish to communicate securely. They can be thought of as complex digital handshakes. Before data is sent, devices must agree on a common language, prove their identities (*authentication*), and establish a secret code to scramble their conversation (*confidentiality*). They are the bedrock upon which our digital trust is built and they are everywhere around us, including:

2: Also called *security protocols.*

- When your phone connects to a cell tower, the **3G**, **4G**, or **5G Authentication and Key Agreement (AKA)** protocol ensures the network knows whom to bill and that your calls or data are not intercepted by a fake base station (*i.e.,* antenna).
- When browsing the web, the **Transport Layer Security (TLS)** protocol has just successfully negotiated a secure channel with the web server, protecting your browsing information from eavesdroppers.
- When you tap your card or smartphone to pay for coffee, the **Europay Mastercard Visa (EMV)** protocol executes a rapid exchange of cryptographic proofs to authorize the transaction with your bank on your behalf.
- When you send a message on Signal or WhatsApp, **Signal**-based end-to-end encryption is designed so that only you and your recipient —not even the service provider— can read the content.
- Finally, while you may never interact with it directly, the **Open Platform Communication Unified Architecture (OPC UA)** protocol is the invisible backbone coordinating the power plants and factories that deliver the electricity and infrastructure all these devices depend on, securing our industrial systems against sabotage.

These protocols derive complex security properties, including confidentiality, authentication, and integrity, from basic mathematical building blocks called *cryptographic primitives* (*e.g.,* encryption schemes or digital signatures). Protocol security does not arise from these primitives in isolation, but from their precise composition into a delicate equilibrium that we picture as a Calder mobile (Figure 1.1): displacing even a single component can destabilize the entire structure. Put differently, if primitives are the bricks —providing mathematically precise guarantees such as "this ciphertext cannot be decrypted without the key"— then protocols are the architectural blueprints that arrange these bricks into a building; yet even a blueprint built from sound bricks can produce a structure that collapses if the foundation is weak or the walls are misaligned. A cryptographic protocol can similarly fail even if its primitives are mathematically



Figure 1.1.: *Untitled* — Calder, 1942. A Calder mobile: each component is delicately balanced and connected to the others by slender wires. The whole structure remains stable only as long as these relations are preserved; displacing a single element can set the entire system off balance. Cryptographic protocol design is ultimately a balancing act: security depends on carefully orchestrating messages, keys, and agents, and a single design flaw can compromise the guarantees of the whole system.

unbreakable. This is the domain of *protocol security*: ensuring that the sequence, structure, and computation of messages actually deliver the promised security goals, even in the presence of malicious actors.

This is where both analogies end. While a building must primarily be robust and safe —withstanding gravity and weather under normal conditions— cryptographic protocols face a far more hostile reality. Beyond random failures, they must remain *secure* in an *adversarial environment*: an active attacker can intercept, modify, and inject messages on the network, and participate in the protocol execution as a malicious agent, while deliberately trying to break the security properties. Such an adversary only has to find a single flaw in the protocol —the weakest link— to break the security of the entire system, while the designers must account for all possible combinations of user and adversary behaviors to ensure security. Achieving this level of security —resistance against an intelligent agent doing everything in their power to subvert the system— is a significantly harder challenge than achieving mere *safety* or *robustness*.

The complexity of these systems —spanning abstract specifications, complex software implementations, and diverse use cases in an adversarial environment— creates a vast attack surface. The history of computer security is a stark reminder of how fragile these protocols are: critical vulnerabilities are uncovered with disconcerting regularity in both the *design* and the *implementation* of cryptographic protocols, often with severe consequences. Design-level flaws —affecting all compliant implementations— and implementation-level bugs —affecting the vulnerable code— can each undermine an entire system, exposing users to risks ranging from data theft and impersonation to state-level surveillance and financial loss. For

instance, the implementation bug Heartbleed [Meh+14], the design-level attack KRACK [VP17], and the long series of attacks against TLS, EMV, and OPC UA —discussed in the next section— have repeatedly demonstrated that even the most widely deployed and scrutinized protocols are not immune to failure. This constant tension between our increasing reliance on digital systems and their inherent fragility motivates the central theme of this manuscript:

> *How can we obtain higher assurance for the cryptographic protocols that underpin our digital world?*
>
> *This manuscript presents a selection of our post-PhD research addressing this question along three complementary axes: (i) the development of fundamental formal methods, (ii) the security analysis of complex real-world protocols, (iii) and the design of techniques to automatically find implementation-level vulnerabilities.*

While the pursuit of higher assurance is a vital goal, it must be balanced by a recognition of the intrinsic limitations of any security analysis. Every proof operates under specific assumptions, notably a *threat model* —the set of capabilities attributed to the potential adversary—; consequently, absolute security remains an illusion, even when vendor claims suggest otherwise. This realization is essential for users before trusting a system, for decision-makers before adopting a technology, and for researchers before declaring a protocol "secure".

1.1. When Cryptographic Protocols Fail

A prominent class of cryptographic protocol failures and vulnerabilities arises from flaws in the *logic* of the protocols themselves. Such *logical attacks* exploit the protocol's logic (as designed or implemented) rather than weaknesses in the underlying cryptographic primitives, *e.g.,* a weak encryption algorithm leaking some bits of the plaintexts. Logical attacks manipulate the sequence and content of messages (by an active network adversary), exploit protocol states, exploit compromised agents, or abuse the interactions between different components to subvert security goals such as confidentiality, integrity, and authentication. We can broadly categorize logical attacks into two main types: *design-level logical attacks* and *implementation-level logical attacks*. We illustrate these types of attacks in Figure 1.2 with examples from the TLS protocol.

Design Flaws Designing protocols is notoriously error-prone as they must remain secure under all interleavings of user and adversary behaviors that the threat model admits. In particular, adversaries can creatively adopt various strategies such as Machine-in-the-Middle (MIM), replay, relay, downgrade, and others. As a result, protocol specifications are often flawed and exposed to *design-level logical attacks*.[3]

In such cases, all compliant implementations are vulnerable to the attack, and the only solution is to fix the design itself. For instance, TLS 1.2 and early drafts of TLS 1.3 suffered from serious design flaws which defeated some of their central goals such as confidentiality and authentication [CVE09-3555; BBK17; Cre+17; Cre+16; Bha+13; MDK14; Bha+14; Avi+16; DG19]. In 2017, the protocol responsible for securing wireless computer networks, WPA2, was shown to be entirely broken by a logical attack known as KRACK [VP17]. Mobile telephony also suffers from such

3: As a concrete illustration of a logical attack, consider the EMV PIN bypass attack of Basin *et al.* [BST21a]. An attacker who can get physically close to a victim's contactless Visa card (*e.g.,* in a crowded place) relays a transaction between the card and an honest, possibly distant, merchant terminal, using two phones connected over the Internet: one plays the terminal to the card, the other plays the card to the terminal. Asked for a payment above the contactless limit, the card honestly answers that online PIN verification is required. The attacker rewrites a single field of this answer, the *Card Transaction Qualifiers*, so that the terminal instead believes that the cardholder was already verified on their own mobile device (*Consumer Device Cardholder Verification Method*). Crucially, no cryptographic primitive is broken: this field is covered by neither the card's cryptogram nor, in this configuration, its signature, so neither the terminal nor the issuing bank can detect the modification. The attacker only needs control over the insecure contactless channel, to relay, intercept, modify, and inject messages. As a result, an attacker can pay amounts above the contactless limit with a victim's card without ever knowing its PIN, defeating one of the core security goals of EMV. Since the flaw lies in the specification itself rather than in a particular implementation, it affects every compliant terminal; Basin *et al.* proposed fixes that can be deployed on terminals and verified them with Tamarin. We present various other design-level logical attacks in detail throughout this manuscript.

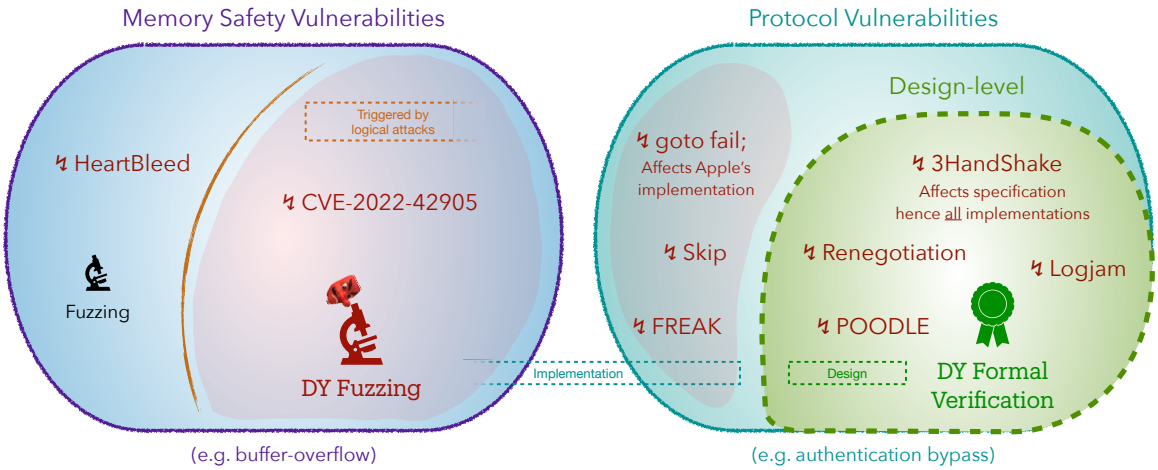


Figure 1.2.: Taxonomy of vulnerabilities in cryptographic protocol specifications and implementations, illustrated with TLS examples (detailed in Section 1.1). *Design-level logical attacks (green area)* exploit flaws in the protocol's logic specification (*e.g.,* Triple Handshake [Bha+14], POODLE [MDK14], Renegotiation [CVE09-3555], Logjam [Adr+15]) and are well-captured by formal verification. Logical attacks can also be introduced at the implementation level through bugs in the code. These can manifest as *protocol vulnerabilities* (such as the authentication bypass caused by Apple's Goto Fail bug [CVE14-1266], or Skip/FREAK [Beu+15]) or as *memory-safety vulnerabilities* (*e.g.,* CVE-2022-42905, see Part C). As we shall see, standard fuzzing targets non-logical memory-safety bugs (*e.g.,* Heartbleed [Meh+14]), but misses the first three classes. The middle area in red highlights the blind spot of current software testing techniques.

attacks: for instance, we showed that 5G-capable phones were exposed to unauthorized tracking and activity monitoring due to a logical flaw in the handling of an internal counter [*Bor+19*] (see Part B). Even credit card payment systems like EMV suffer from such flaws, as evidenced by recent PIN bypass attacks [BST21a; BST21b; BST23].

Implementation Flaws Protocol design specifications are often just a blueprint for the programs that end-users actually run, which are frequently plagued with bugs that were non-existent in the specification. These implementation-level flaws can be broadly categorized into two classes (we discuss other types of attacks later):

- *Implementation-level protocol vulnerabilities*: These occur when bugs in the code introduce new logical vulnerabilities absent from the specification. Examples include authentication bypasses (*e.g.,* Apple's "Goto Fail" bug[4]), state-machine violations (*e.g.,* the FREAK attack [Beu+15]), or improper certificate validation [RP15].
- *(Memory) safety vulnerabilities*: These are caused by non-protocol-specific software bugs such as software faults (deadlock, stack overflow, division by zero, *etc.*) or spatial and temporal memory bugs, such as buffer overflows, over-reads (*e.g.,* the famous Heartbleed bug [Meh+14]), or memory leaks. They can manifest as simple system crashes or lead to severe security breaches such as private key leakage or Remote Code Execution (RCE). Importantly, the boundary with logical attacks is porous: some memory-safety vulnerabilities require a specific sequence and content of adversarial messages to be triggered, and are therefore best understood as logical attacks that happen to manifest as memory-safety violations[5]. Others are independent of the protocol logic (*e.g.,* a simple out-of-bounds read in a parsing function) and are traditionally targeted by standard fuzzing techniques (see Section 1.2.3).

Other types of attacks Finally, it is worth mentioning that cryptographic protocols can also be compromised by *attacks on the crypto-*

4: Apple's "Goto Fail" bug (CVE-2014-1266 [CVE14-1266]) is a code-level flaw that silently disables signature verification along a specific TLS handshake path, allowing a network attacker to impersonate any web server to a vulnerable client. As with all implementation-level protocol vulnerabilities, finding such bugs requires (i) steering the handshake through a carefully crafted sequence of adversarial messages to reach the buggy code path, and (ii) being able to detect a protocol vulnerability —here an authentication bypass— rather than a mere crash or memory corruption, which can be significantly harder. See sidenote 24 in Part C for a more detailed presentation.

5: For instance, we discovered a stack buffer overflow (CVE-2022-39173) and a heap buffer over-read (CVE-2022-42905) in a prominent TLS implementation that are reachable only through specific sequences of handshake messages. In particular, the former arises because the server fails to revert side effects during a failed cipher list negotiation triggered by adversarially crafted messages. An attacker can exploit this by crafting a complex sequence of handshake messages, ultimately triggering a massive stack buffer overflow of up to 44,700 bytes (see Part C).

graphic primitives (*e.g.,* cryptanalysis [Bei+21] or backdoored implementations [Tho84]), *side-channel attacks* (exploiting physical leakage such as timing or power [MOP07]), *micro-architectural attacks* (see SoK [Can+19]), *physical attacks* (hardware tampering), or *social engineering* (manipulating users). While these are critical threats, they are distinct from logical attacks, and fall outside this manuscript's scope. However, the distinction is not always clear-cut. For instance, in Chapter 3, we incorporate known primitive weaknesses (*e.g.,* weak hash functions) into our symbolic models to capture attacks exploiting the interplay between protocol logic and primitive-level weaknesses. Furthermore, while the fuzzing techniques of Part C primarily target logical attacks in protocol implementations, they can also, *en passant*, detect memory-safety violations.

1.2. State of the Art in Cryptographic Protocol Analysis

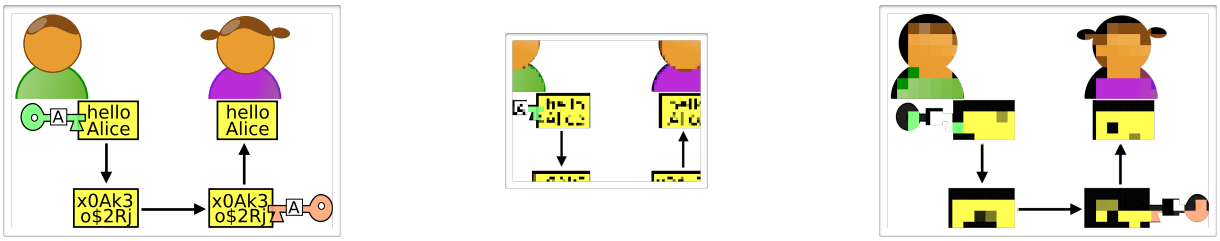
Improving assurance for cryptographic protocols has been a decades-long research program, pursued along several complementary directions. The distinctions drawn in the preceding section —between design-level and implementation-level attacks— map onto the scope of each class of technique. Design-level flaws are typically addressed through *symbolic* and *computational security verification*, while implementation-level flaws are dealt with using *testing* (including *fuzzing*) and *program verification*.

1.2.1. Design-level Verification

Formal methods offer rigorous, mathematical frameworks and techniques to analyze cryptographic protocol specifications, hence at the design-level. Two main distinct approaches have emerged since the 80s: the *computational approach* and the *symbolic approach*. As manually establishing these proofs is expertise-demanding, tedious, and highly error-prone, approaches have been proposed where security-oriented proofs are machine-checked or even automated (*i.e.,* machine-constructed). Those techniques are often referred to as *Computer-Aided Cryptography (CAC)*; see [Bar+21a] for a survey. Given the complexity of real-world protocols, proof automation will be a key aspect of these techniques.

[Bar+21a]: Barbosa, Barthe, Bhargavan, et al., *SoK: Computer-Aided Cryptography*, S&P'21

Computational security The computational approach consists in analyzing cryptographic protocols and primitive specifications in the *computational model*: exchanged messages are modeled as bit strings, while agents and the attacker are modeled as Probabilistic Polynomial-time Turing Machines (PPTMs). Computational security offers strong security guarantees by virtue of its rich attacker model but has long suffered from a lack of automation due to the high complexity of this model. Different techniques and tools have been developed, each making a different trade-off between expressivity and automation [Bar+21a]: EasyCrypt [Bar+11], CryptoVerif [Bla08], Squirrel [Bae+21], SSProve [Has+23], F* [Swa+16], *etc.* While some of these tools provide some level of automation, they remain expertise-demanding and do not scale well to whole real-world protocols. A single person-year was required for the EasyCrypt proofs in [Cor+17]; similarly, while the TLS 1.3 sub-protocols were each proved in CryptoVerif with manual guidance, the core composition proof tying them together had to be done by hand on paper [BBK17]. Verification in



(a) Real-world protocol **(b)** Computational model **(c)** Symbolic model

Figure 1.3.: Intuitively visualizing the trade-off between computational and symbolic models for a simple encrypted channel (Bob sends an encrypted message under a shared key). The *computational model* (b) retains the bit-string reality but abstracts it into mathematical frameworks of PPTMs; the complexity of this model limits automation and scalability and thus, in practice, the scope of analyses. The *symbolic model* (c) abstracts further by treating encryption as an idealized operation, losing precision at the primitive level but making whole-protocol automation more tractable. Both perspectives are complementary and necessary for comprehensive security assurance.

F* also demands substantial effort, *e.g.,* approximately 20 person-months for the Quick UDP Internet Connections (QUIC) record layer [Del+21].[6]

6: Albeit covering actual implementations rather than mere protocol specifications, see Section 1.2.2.

Symbolic security To automatically analyze complex protocols —rather than isolated primitives or sub-protocols[7]— and to explore a variety of threat models and security properties, we turn to a more abstract model for which proof automation is easier and scales better (see Figure 1.3). The model that is now widely adopted in the field of protocol analysis is the *symbolic model* (or Dolev-Yao (DY) model), pioneered by Dolev and Yao in the early 1980s [DY83].

7: For instance, the Quick UDP Internet Connections (QUIC) or TLS record layer protocols are each only a component of a much larger protocol that also includes a complex handshake protocol; such full protocols have been analyzed end-to-end using the symbolic approach [Cre+17; BBK17].

[DY83]: Dolev and Yao, *On the security of public key protocols*, TIT'83

This model is designed to capture logical attacks at the design level and will thus be our formal model of choice. This mathematical model makes strong assumptions on cryptographic primitives by considering them as black-box functions satisfying some algebraic relations in a *term algebra* (*e.g.,* signature forgery is impossible without the full signature key). Honest protocol agents are typically modeled in process calculi (*e.g.,* applied π-calculus) or rule-based formalisms (*e.g.,* multiset rewriting rules) and interact with a powerful attacker who has complete control over the network and whose deduction capabilities are defined by the term algebra. Security properties such as confidentiality or authentication are typically expressed as *reachability properties*, *i.e.,* first-order formulas over exchanged messages and execution steps. Privacy properties such as anonymity or unlinkability are typically expressed as *behavioral equivalence properties* between two systems, such as *trace equivalence*. The symbolic model offers various mathematical techniques to formally reason about protocols, ranging from *rewriting theory* to *resolution in logic* and *model-checking* that benefit from mature and efficient machine support with tools such as ProVerif [Bla16], Tamarin [Mei+13], DeepSec [CKR18], Sapic+ [Che+22], *etc.* This places this model in a sweet spot between automation and precision for analyzing real-world protocols. We present this symbolic model and the associated verification tools in greater detail in Chapter 2.

Note on Post-Quantum Security *Post-Quantum (PQ) security* refers to the resistance against adversaries equipped with quantum computers, which could break widely deployed primitives based on integer factorization and discrete logarithm [BBD09; Che+16]. In response to this threat, protocols are evolving to integrate quantum-resistant primitives, such as those standardized by NIST (*e.g.,* ML-KEM and ML-DSA), sometimes through *hybrid constructions* that combine classical and post-quantum algorithms [GCR23]. Notable examples include the PQXDH protocol for

[BBD09]: Bernstein, Buchmann, and Dahmen, *Post-Quantum Cryptography*, 2009

[Che+16]: Chen, Jordan, Liu, et al., *Report on Post-Quantum Cryptography*, 2016

secure messaging [Bha+24a] and hybrid versions of WireGuard [Laf+25]. Formal security proofs must now account for such post-quantum attackers as security agencies recommend transitioning to PQ-resistant protocols [Ame24] or hybrid security [Fre26]. In the computational model, this involves developing new logics and tools that are sound against quantum adversaries [Bar+21b; CFJ22]. Because the symbolic model treats cryptographic primitives as black-box term-algebra operators, its core proof techniques transfer to the post-quantum setting —provided the new primitives' algebraic properties are explicitly modeled. However, analyzing the transition to PQ security in this model still requires accounting for new primitive interactions and hybrid constructions, as demonstrated by several recent symbolic analyses using ProVerif and Tamarin for protocols like KEMTLS [Cel+22], PQXDH [Bha+24a], and hybrid Wireguard [Laf+25; LMR24].

[Ame24]: American National Institute of Standards and Technology (NIST), USA, *Transition to Post-Quantum Cryptography Standards*, 2024

[Fre26]: French National Agency for the Security of Information Systems (ANSSI), *Rules and Recommendations Concerning the Choice and Sizing of Cryptographic Mechanisms*, 2026

1.2.2. Implementation-level Verification

While design-level symbolic verification has reached significant maturity for analyzing protocol specifications, it provides no guarantees for the code that actually runs on users' devices. Closing this gap is notably the goal of *program verification*. Historically, it primarily focused on checking functional correctness and safety properties (*e.g.,* memory safety) in programs. Since the mid-2000s, more security-oriented program verification techniques have emerged, but were initially limited in scope and real-world applicability [APS14]. In the last decade, the field has seen significant advances, including the first real-world applications of these techniques, which we briefly summarize below.

[APS14]: Avalle, Pironti, and Sisto, *Formal verification of security protocol implementations: a survey*, FAC'14

Clean-slate verification Project Everest pioneered the first large-scale application of clean-slate verification to real-world protocols, delivering verified TLS and QUIC stacks together with the HACL* cryptographic library [Ahm+26; Zin+17].[8] More generally, beyond cryptographic primitives, two complementary strategies establish a sound link between protocol models and executable code. *(a) Code generation from verified specifications* compiles a model into an implementation whose behavior provably refines that of the model: examples include OCaml from CryptoVerif, Rust from Owl, Java/Go from Tamarin, *etc.* [CB13a; CB15; SGP25; Spr+20; Arq+23; Arq+26; Mod16; Sis+18; AMV15; GMF25]. *(b) Model extraction from existing code* goes in the opposite direction, recovering a verifiable specification (or directly verifying the code) from a given implementation, with sources ranging from F#, JavaScript, C, and Rust to compiled binaries, *etc.* and back-ends including ProVerif, CryptoVerif, F*/DY*, and general-purpose program verifiers [BCF07; Bha+08; BFK16; BBK17; KBB17; AGJ11; AGJ12; NKN23; VJ15; Dup+14; Swa+11; Bha+21; Wal+23; Bha+24b; Bha+25]. A recent end-to-end illustration is Bertie, a Rust post-quantum TLS 1.3 prototype verified via hax combining F* (runtime safety), ProVerif (protocol security), and SSProve (cryptographic constructions), on top of libcrux [Bha+25].

8: HACL* provides verified, high-performance cryptographic *primitives* (not protocol implementations) now shipped in the Linux kernel and Mozilla Firefox; dedicated toolchains such as Vale and Jasmin push this further to machine-checked assembly-level code [Bon+17; Alm+20].

Scalability and legacy challenges Despite such advances, two fundamental challenges persist. First, even for clean-slate proofs, the required expert effort remains prohibitive: We already mentioned the 20 person-months verification of the QUIC record layer sub-protocol [Del+21]. Verifying two of the three sub-protocols of the Messaging Layer Security (MLS) protocol in DY* required about *two person-years* of expert

effort [Wal+23]. Even with refinement-based frameworks like Igloo [Spr+20] or its extension [Arq+23], the proof-to-code ratio routinely exceeds practical limits for large codebases (3:1 for [Arq+23]). Model-extraction techniques face the same scalability challenges: *e.g.,* Bertie has a proof-to-code ratio of nearly 15:1 [Bha+25]. The recent Diodon methodology [Arq+26] focuses the verification effort on the protocol code, discharging the surrounding application layer via automatic static analyses. Yet, it still carries a substantial proof-to-code ratio on the protocol code (4:1), and large or complex implementations remain out of reach.[9]

9: Lighter-weight alternatives forgo soundness for scalability (in that the code is not fully verified): model extraction from binaries [NKN23] or runtime monitoring against Tamarin specifications [MK24] apply to legacy code but still struggle with the complexity and non-determinism of large real-world systems.

Crucially, these clean-slate approaches are largely inapplicable to widely deployed codebases: prominent TLS implementations such as OpenSSL, boringSSL, wolfSSL, and LibreSSL each comprise hundreds of thousands to over a million lines of C/C++ relying on low-level features (*e.g.,* pointer arithmetic) that are currently beyond the reach of automated program verifiers, leaving these widely deployed systems exposed to both protocol and memory-safety bugs.

As a result, a gap remains between verifying the design of complex protocols and analyzing their full, real-world implementations. This limitation motivates the use of more pragmatic, automated techniques such as testing and fuzzing, which we discuss in the next section.

1.2.3. Testing and Fuzzing

Security-oriented program verification is generally too costly and expertise-demanding to integrate into practitioners' coding workflows. Instead, they extensively rely on *testing* to augment their confidence that their programs are secure. As opposed to formal verification, testing does not aim to obtain sound proofs, *i.e.,* attacks may be missed. It is a pragmatic approach: by covering a broad range of behaviors, a test suite provides confidence proportional to its *coverage* —the fraction of code paths exercised. Achieving high coverage of the Program Under Test (PUT) is therefore essential. Classical random testing falls short when the PUT contains strict path conditions —typically when messages are integrity-protected or encrypted— as is the case for security-oriented programs. One of the gold standards of software testing that remedies this problem is *fuzz testing* (or fuzzing for short), see the surveys [Man+21; Zhu+22]. It was first proposed in the 1990s and is now an integral component of software development practices. It is widely used by practitioners —Google [Bab+19], Adobe, Cisco, and Microsoft use it at scale on their codebases— and has become a subject of extensive academic research.

[Man+21]: Manès, Han, Han, et al., *The Art, Science, and Engineering of Fuzzing: A Survey*, TSE'21
[Zhu+22]: Zhu, Wen, Camtepe, et al., *Fuzzing: A Survey for Roadmap*, ACM Comput. Surv.'22

In its general form, *fuzzing* is the act of repeatedly executing the PUT on inputs beyond the intended input space to find violations of certain *security policies.* Most of the time, this is done by iterating fuzz runs where new test cases are generated from the current *seed pool* whose execution by the PUT will provide *feedback* to the algorithm that will guide future test case generation. The nature of the feedback can be diverse: coverage (*e.g., code coverage*), policy violation, *etc.* Different approaches exist for input generation [Man+21]: *generation-based fuzzers* utilize a specification of the input space (*e.g.,* a grammar) while *mutation-based fuzzers* leverage (often random) mutations on a corpus of valid executions (*e.g.,* using evolutionary algorithms).

Narrowing down the discussion to cryptographic protocols, the main approach is mutation-based fuzzing on the network packets or the inputs of cryptographic primitives. For example, the Cryptofuzz project [Vra20] has developed bit-level, mutation-based fuzzers for various symmetric

cryptographic primitives from many different libraries. Such testing methodologies are adequate to find some safety vulnerabilities with potential security implications. However, they are generally unable to find logical attacks —whether these manifest as memory-safety or protocol vulnerabilities— because random bit-level mutations such as bit flips on network packets almost never emulate the structured operations of a Dolev-Yao (DY) attacker. Such operations include performing cryptographic operations, MIM attacks, replay attacks, credential swapping, or downgrade attacks.[10] Moreover, in addition to this *reachability problem*, these fuzzers also face a *detection problem* as they are generally limited to detecting safety bugs and are unable to detect protocol vulnerabilities[11]; the latter would simply be silently ignored.

In Part C, we show that neither bit-level fuzzers nor more structured protocol fuzzers can capture the DY attacker's capabilities and therefore miss implementation-level logical attacks that such an attacker could exploit.

10: Said differently, these fuzzers fail to capture the capabilities of a *DY attacker* and the logical attacks that such an attacker can mount. The vulnerabilities missed by these fuzzers correspond to the whole area to the right of the orange curve ("Triggered by logical attacks") in Figure 1.2. That said and as previously mentioned, programs may suffer from memory-safety vulnerabilities that do not require logical attacks to be triggered (*e.g.,* Heartbleed [Meh+14]), for which bit-level fuzzing is adequate.

11: See the teal area "Protocol Vulnerabilities" in the right half of Figure 1.2.

1.3. Research Journey Overview

Around 2017, at the conclusion of our doctoral studies [*Hir17*], the field of protocol security was at a pivotal juncture: the first large-scale formal analyses of deployed standards were demonstrating that automated verification tools could engage with real-world protocols and standardization processes, as exemplified by the *Reactive and Proactive Standardisation of TLS* [PM16]. Following this momentum, we began analyzing other large-scale standards, *e.g.,* 5G AKA and later OPC UA.

However, as formal analyses became more ambitious, new challenges came into sharp focus, such as the need for more realistic cryptographic primitive models. Moreover, a simple binary "secure/insecure" verdict became insufficient; we lacked the methodology to *measure* security across a landscape of different threat models and properties, or in the presence of weak cryptographic primitives. Consequently, our research journey became defined by a continuous feedback loop: tackling large-scale, real-world security analyses (**Part B**) drove the development of the foundational modeling techniques, verification algorithms, and proof methodologies required to make those analyses possible (**Part A**).

Even with improved models, a more fundamental limitation remained: the persistent gap between the symbolic verification of designs and the security of real-world implementations. Our large-scale analyses frequently led us to manually inspect protocol implementations, sometimes uncovering implementation-level logical attacks —such as the privacy and election integrity vulnerabilities we found in the French Legislative E-Voting Protocol (FLEP) (discussed in **Part B**). Recognizing that these implementation-level logical attacks remained in a blind spot both[12] for standard design-level verification and fuzzing, we defined a project that we have pursued ever since: *DY model-guided fuzzing* (**Part C**).

12: Red area in the middle of Figure 1.2.

The next three subsections summarize each part; detailed contribution overviews appear in the introductions to Parts A to C. Figure 1.4 maps this journey and shows how a selection of our works is positioned and connected.

We cite our own works in *dark blue italics*.

[*Hir17*]: Hirschi, *Automated Verification of Privacy in Security Protocols: Back and Forth Between Theory & Practice (PhD thesis)*, 2017

[PM16]: Paterson and Merwe, *Reactive and Proactive Standardisation of TLS*, SSR'16

1.3.1. Part A: Advancing Formal Methods for Cryptographic Protocol Security Verification

Gaps A first set of contributions concerns the development of formal methods themselves. Several of these contributions were prompted by concrete obstacles encountered during the large-scale analyses of Part B, where state-of-the-art tools could not be used out of the box. We identify three recurring gaps in the formal verification of cryptographic protocols. The first is a *modeling* gap: in some cases, the standard symbolic abstractions of cryptographic primitives are too coarse to capture realistic attacks, and need to be refined —for instance, to account for known weaknesses of hash functions or for the algebraic properties of Exclusive OR (XOR). The second is a *scalability and expressiveness* gap for *verification algorithms*: equivalence-based properties —used to model some key privacy properties— and richer models answering the first gap (such as XOR) are notoriously hard for verification engines, and existing tools often fail to scale or even to apply. The third is a *methodological* gap: with the increasing complexity of protocols and their analyses, the traditional binary verdict of *secure* vs. *insecure* is no longer informative enough. In particular, it does not define the limits of the achieved security, and it does not characterize the conditions under which security holds.

Research questions The corresponding research questions structure Part A:

- *How should symbolic models be refined to capture relevant properties of real-world primitives while remaining tractable for automated verification?*
- *How can verification algorithms be made effective to analyze equivalence-based properties?*
- *How can verification be organized to* measure *security rather than to deliver a single binary verdict?*

Key contributions On the modeling side, we introduce techniques to incorporate weakened hash functions into symbolic models, exposing logical attacks that exploit primitive-level weaknesses (Chapter 3) [*Che+23*]; we also develop dedicated abstractions for identity-based and distributed cryptographic protocols [*BHS19*; *HSB21*].

On the algorithmic and expressiveness side, we develop more general partial-order reduction techniques for equivalence properties with dramatic speed-ups [*BDH17*], and sound sufficient conditions that reduce privacy properties in e-voting to reachability and thereby enlarge the class of protocols within reach of existing tools [*HC19*]. We also extend symbolic verification to protocols using XOR [*Dre+18*; *Dre+20*] or purely associative operators [*Che+23*] —an extension motivated by our analyses of 4G and 5G AKA [*Bas+18*; *Bor+19*] and of hash-based attacks, now integrated into Tamarin.

On the methodological side, we propose *lattice exploration*: threat models and security goals (and possibly protocol variants or protocol features) are organized into lattices that are then automatically explored, producing a fine-grained map of the security limits of a protocol [*Gir+20*]. This methodology was a decisive enabler for many of our analyses [*Che+23*; *DHK25a*], and is, beyond these case studies, a general way of using formal methods to *measure* security rather than only to assert it for an a priori fixed set of assumptions and goals.

[*Che+23*]: Cheval, Cremers, Dax, et al., *Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses*, USENIX'23
[*BHS19*]: Basin, Hirschi, and Sasse, *Symbolic Analysis of Identity-Based Protocols*, Meadows Festschrift'19
[*HSB21*]: Hirschi, Schmid, and Basin, *Fixing the Achilles Heel of E-Voting: The Bulletin Board*, CSF'21
[*BDH17*]: Baelde, Delaune, and Hirschi, *A Reduced Semantics for Deciding Trace Equivalence*, LMCS'17
[*HC19*]: Hirschi and Cremers, *Improving Automated Symbolic Analysis of Ballot Secrecy for E-Voting Protocols: A Method Based on Sufficient Conditions*, EuroS&P'19
[*Dre+18*]: Dreier, Hirschi, Radomirovic, et al., *Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR*, CSF'18
[*Dre+20*]: Dreier, Hirschi, Radomirovic, et al., *Verification of stateful cryptographic protocols with exclusive OR*, JCS'20
[*Gir+20*]: Girol, Hirschi, Sasse, et al., *A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols*, USENIX'20

1.3.2. Part B: Large-Scale Real-World Protocol Analyses

Gaps A second strand of our work concerns the application of formal methods to protocols that are actually in operational use: mobile telephony, Industrial Control Systems (ICS), and national elections. Such systems differ from academic case studies in several respects. Their specifications are typically long, informal, and distributed across multiple documents that describe components in isolation; threat models and security goals are rarely stated explicitly; and, in some cases no complete public specification is available at all, so that the protocol must first be reverse-engineered from a deployed system. Because these protocols evolve through standardization processes that are not aligned with academic notions of proof, the impact of a formal analysis depends not only on its scientific content but also on the way it is conducted, communicated, and reported.

Research questions The questions we address in Part B are deliberately practical:

> - *Can formal methods uncover previously unknown vulnerabilities in widely deployed standards that have already been the subject of substantial expert review?*
> - *Can such analyses produce concrete fixes that standardization bodies and stakeholders are willing to adopt, and ideally fixes that come with formal guarantees?*
> - *What general lessons —for the protocols, the designers, the formal methods' community, and the research community at large— can be extracted from these case studies, beyond the discovery of individual flaws?*

Key contributions We address these questions through three case studies.

- **4G/5G AKA:** We developed a formal model of this international mobile communication standard and identified logical attacks that weaken authentication and introduce severe privacy leaks, enabling subscriber tracking and monitoring (*e.g.,* via sequence number tracking) in 4G and 5G [*Bas+18*; *Bor+19*]. We proposed provably secure fixes and responsibly disclosed our findings to the 3rd Generation Partnership Project (3GPP). Our analysis eventually led to changes in the 5G standard to prevent the privacy attacks we identified.
- **OPC UA:** We analyzed this widely deployed ICS protocol, often described as "secure-by-design", and built one of the largest ProVerif protocol models to date [*DHK25a*]. Our analysis exposed various logical attacks, including client and user impersonation, and contributed to a major revision of the standard following a long responsible disclosure process with the OPC UA Foundation, which assigned 8 Global CVE Allocation System (GCVE) identifiers to the vulnerabilities we discovered.
- **E-Voting:** We then analyzed the French Legislative E-Voting Protocol (FLEP) that is used for legislative elections in France since 2022 [*DH23*]. We reverse-engineered the deployed system and found verifiability and ballot-secrecy logical attacks —one at the design level and one at the implementation level— that defeat the core security properties of the system, *i.e.,* election integrity and voter

[*Bas+18*]: Basin, Dreier, Hirschi, et al., *A Formal Analysis of 5G Authentication*, CCS'18
[*Bor+19*]: Borgaonkar, Hirschi, Park, et al., *New Privacy Threat on 3G, 4G, and Upcoming 5G AKA Protocols*, PoPETS'19

[*DHK25a*]: Diemunsch, Hirschi, and Kremer, *A Comprehensive Formal Security Analysis of OPC UA*, USENIX'25

[*DH23*]: Debant and Hirschi, *Reversing, Breaking, and Fixing the French Legislative Election E-Voting Protocol*, USENIX'23

privacy.[13] After a long responsible disclosure process and discussion about ways to fix these vulnerabilities, the protocol was updated for the 2023 and 2024 elections.

13: This work was awarded the CNIL – Inria Data Protection 2023 prize.

Beyond these specific results, Part II contributes: *(i)* formal models of opaque or sprawling specifications that serve as reusable artifacts; *(ii)* methodological reflections on interfacing formal verification with standardization bodies; and *(iii)* broader lessons for improving both the design of such protocols and the application of formal methods to their analysis.

1.3.3. Part C: Formal Models Meet Fuzz Testing

Gaps The last strand of our work addresses a limitation that is shared by all the analyses of Parts A and B: the gap between a verified design and the code that actually runs. A specification can be formally proven secure while its implementations exhibit logical attacks —authentication bypasses, state-machine deviations, missing checks— that lie outside the scope of design-level verification. Such implementation-level logical attacks have been discussed earlier in this introduction (*e.g., Goto Fail*, *FREAK*); we have ourselves come across instances of this class while inspecting code in the context of our aforementioned FLEP analysis. Standard fuzzing, the dominant technique for finding bugs in protocol implementations, is not designed for this class of attacks: its mutations operate on bit strings rather than on cryptographically meaningful messages and fall short of the DY attacker's capabilities; its bug detection engine is further limited to memory-safety errors, causing logical attacks that do not crash the program to be silently missed.

Research questions Part C therefore investigates the following questions:

- *Can fuzzing efficiently explore the blind spot of implementation-level logical attacks?*
- *Can the DY attacker model be integrated into a fuzzing loop to find logical attacks? Could it detect protocol-level violations, not only memory-safety errors?*
- *Does the resulting approach scale to large codebases such as mainstream TLS implementations, and does it detect attacks that are missed by state-of-the-art fuzzers?*

Key contributions We propose a new fuzzing approach that we call *DY model-guided fuzzing* (*DY fuzzing* or DYF for short) [*AHK24*]. The key idea is to use the DY model as a guide for fuzzing protocol implementations. Test cases are formal *DY traces*: structured executions of a protocol against a DY attacker, in which messages are represented as formal terms and adversarial actions are symbolic operations (composition, decomposition, re-encryption, replay, injection). We design term-level fuzzing mutations on traces; we define a concretization procedure that compiles each DY trace into a sequence of network interactions with the PUT; and we derive security oracles from formal security goals, so that protocol-level violations can be detected even in the absence of safety errors.

[*AHK24*]: Ammann, Hirschi, and Kremer, *DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing*, S&P'24

We also proposed a differential-testing extension to DY fuzzing that we dubbed Differential Dolev-Yao Fuzzing (DDYF) [*GHK27*]. Its new security oracle identifies semantic behavior differences between two protocol

[*GHK27*]: Gouville, Hirschi, and Kremer, *DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols*, S&P'27

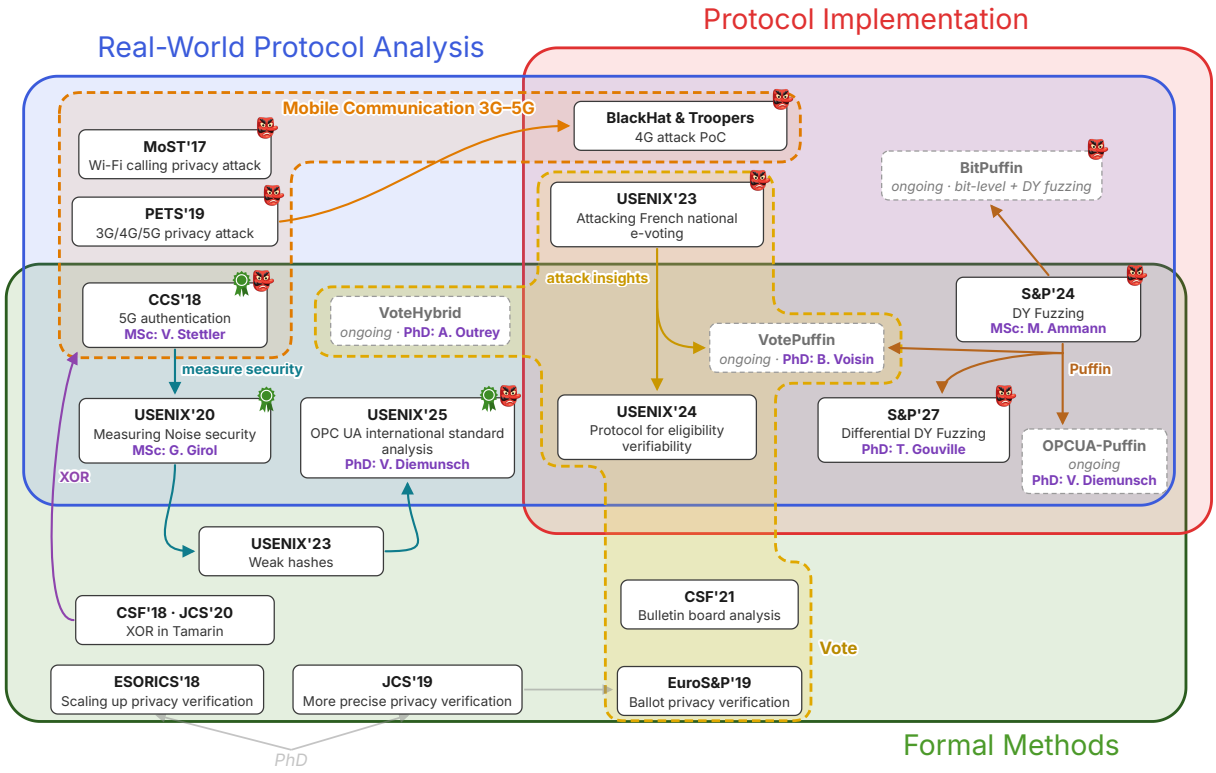


Figure 1.4.: Research Journey Overview: selection of works and contributions highlighted in this manuscript. Symbol 😈 indicates an analysis that uncovered new attacks on real-world protocols, acknowledged by vendors; 🏅 refers to large-scale formal analyses of standards; PhD MSc indicates a work accomplished with students we supervised (their names are shown in purple).

implementations by re-interpreting execution discrepancies through the DY attacker model. After triaging, a difference may indicate a bug or vulnerability in one of the implementations.

We implement our DYF and DDYF designs in PUFFIN [puffin], a modular library to build DY fuzzers for arbitrary cryptographic protocols. We also specialize it to the case of TLS, resulting in TLSPUFFIN, a full-fledged DY fuzzer for TLS that we apply to the latest versions of OPENSSL, BORINGSSL, WOLFSSL, and LIBRESSL. We made DYF and PUFFIN modular, which is crucial for reusability, and this modularity allowed us to easily adapt the fuzzer to new protocols such as SSH, as we demonstrate with SSHPUFFIN. The evaluation of PUFFIN yields 5 new Common Vulnerabilities and Exposuress (CVEs) on WOLFSSL—one critical and two high severity, all missed by state-of-the-art standard fuzzers— and, through DDYF, 52 specification-compliance violations, including four security-related bugs, across OPENSSL, BORINGSSL, WOLFSSL, LIBRESSL, WOLFSSH, and LIBSSH.

[puffin]: puffin team, *TLSPUFFIN: a Dolev-Yao model-guided fuzzer for cryptographic protocol implementations:* https://github.com/tlspuffin/tlspuffin

Part C thus connects the two previous parts: the formal models developed for design-level verification become operational guides for testing deployed implementations. While DY fuzzing does not yield formal guarantees, our TLS evaluation demonstrates that it provides an effective means of uncovering implementation-level logical attacks that standard fuzzers miss[14]. The approach also opens a research program of its own, sketched in the perspectives chapter.

14: DYF precisely targets this blind spot (red area in Figure 1.2), covering both memory-safety vulnerabilities triggered by logical attacks —thus addressing the reachability problem— and protocol vulnerabilities —thus also addressing the detection problem; both problems are discussed in Section 1.2.3.

Outline of the Manuscript

A preliminary **Background** chapter (Chapter 2) introduces cryptographic protocols, the Dolev-Yao (DY) model and the symbolic framework shared by all three parts, and the main verification tools we use throughout.

Part A (*Advancing Formal Methods for Cryptographic Protocol Security Verification*) presents our contributions to the modeling, algorithmic, and methodological foundations of symbolic verification.

Part B (*Large-Scale Protocol Analyses*) presents three major real-world case studies: the 5G AKA mobile telephony standard, the OPC UA ICS protocol, and the French Legislative E-Voting Protocol (FLEP).

Part C (*Formal Models Meet Fuzz Testing*) presents our DY model-guided fuzzing approach, its implementation in PUFFIN and TLSPUFFIN, and its evaluation on mainstream TLS stacks.

A concluding chapter (Chapter 11) presents our perspectives for future research.

Collaborators

Research is inherently collaborative. The work presented in this manuscript results from collaborations with colleagues in the international security and verification community. The following list includes all co-authors of publications within the scope of this HDR since the completion of our PhD, together with their institutional affiliations at the time of the relevant work.

- **Max Ammann**, TU Munich, Germany & Trail of Bits, USA
- **David Baelde**, LSV, ENS Paris-Saclay, France
- **David Basin**, ETH Zürich, Switzerland
- **Ravishankar Borgaonkar**, TU Berlin, Germany & SINTEF Digital, Norway & NTNU, Norway
- **Vincent Cheval**, Inria Paris, France
- **Véronique Cortier**, LORIA, CNRS, France
- **Cas Cremers**, CISPA Helmholtz Center for Information Security, Germany & University of Oxford, UK
- **Alexander Dax**, CISPA Helmholtz Center for Information Security, Germany
- **Alexandre Debant**, LORIA, Inria Nancy, France
- **Stéphanie Delaune**, IRISA, Université de Rennes, France
- **Vincent Diemunsch**, ANSSI & LORIA, Inria Nancy, France
- **Jannik Dreier**, LORIA, Université de Lorraine, France
- **Guillaume Girol**, École Polytechnique, France & ETH Zürich, Switzerland & CEA, List, Université Paris-Saclay, France
- **Anselme Goetschmann**, LORIA, Inria Nancy, France
- **Tom Gouville**, LORIA, Inria Nancy, France
- **Dennis Jackson**, University of Oxford, UK
- **Charlie Jacomme**, Inria Paris, France
- **Steve Kremer**, LORIA, Inria Nancy, France
- **Piers O'Hanlon**, University of Oxford, UK
- **Shinjo Park**, TU Berlin, Germany
- **Saša Radomirović**, ETH Zürich, Switzerland & University of Dundee, UK
- **Ralf Sasse**, ETH Zürich, Switzerland
- **Lara Schmid**, ETH Zürich, Switzerland

- **Altaf Shaik**, TU Berlin, Germany
- **Vincent Stettler**, ETH Zürich, Switzerland

Several of these collaborations arose directly from our supervision of students and research engineers, whose work contributed to joint publications.

- **Vincent Stettler** (ETH Zürich, 2017) was a master's student we co-supervised with Ralf Sasse on next-generation mobile network security analysis; he is a co-author of [*Bas+18*].
- **Guillaume Girol** (École Polytechnique & ETH Zürich & CEA, Université Paris-Saclay, 2019) carried out a six-month master's internship under our co-supervision (with Ralf Sasse) on the formal analysis of the Noise framework; he is a co-author of [*Gir+20*].
- **Max Ammann** (TU Munich, 2021) carried out a five-month master's internship under our co-supervision (with Steve Kremer) on DY model-guided fuzzing of cryptographic protocol implementations; he is a co-author of [*AHK24*].
- **Anselme Goetschmann** (Inria Nancy, 2022) was an engineering intern working on the implementation of an OpenID-based eligibility verifiability protocol; he is a co-author of [*Cor+24*].
- **Vincent Diemunsch** (ANSSI & Inria Nancy, 2022–2026) is a PhD student under our co-supervision (with Steve Kremer), whose work focuses on the formal analysis of industrial communication protocols and who will defend his thesis in December 2026; he is a co-author of [*DHK25a*].
- **Tom Gouville** (Inria Nancy, 2023–2027) is a PhD student under our co-supervision (with Steve Kremer), whose work focuses on differential DY fuzzing of cryptographic protocol implementations—he has co-authored [*GHK27*]—and on semi-automating the integration of new protocols into DY fuzzers.

Other research directions described in this manuscript are currently being pursued with PhD students and research engineers whom we co-supervise:

- **Arthur Outrey** (ENS Lyon & Inria Nancy, 2026–) and **Benjamin Voisin** (ENS Rennes & Inria Nancy, 2026–) have just finished their end-of-studies internships with us and are now PhD students under our co-supervision (with Alexandre Debant) on, respectively, the design of hybrid-secure e-voting protocols and the fuzzing of e-voting protocols.
- **Nataël Baffou** (Inria Nancy, 2026–2027), **Olivier Demengeon** (Inria Nancy, 2025–), and **Michael Mera** (Inria Nancy, 2024–2025) are/were research engineers under our supervision, working on engineering support for our DY fuzzer project (PUFFIN).

2. Background

This chapter provides background on some concepts and techniques used in this manuscript. We first present the main concepts and building blocks of cryptographic protocols and discuss some examples of protocols in Sections 2.1 and 2.2. Then, we present the formal model we shall use to represent cryptographic messages, protocols, and security properties in Section 2.3. Finally, we present the main techniques and tools for automated verification of cryptographic protocols in Section 2.4.

2.1 Standard Cryptographic Primitives 18
2.2 Security Protocols 19
2.2.1 Sigma', an Authenticated Key Exchange Protocol . . 20
2.2.2 The Noise Family of AKE Protocols 20
2.3 Dolev-Yao Model 24
2.3.1 Cryptographic Messages Models 24
2.3.2 Protocol Models 25
2.3.3 Security Property Models 27
2.4 Automated Verification 28
2.4.1 The Tamarin Prover . . . 29
2.4.2 The ProVerif Prover . . . 33
2.4.3 Other Approaches and Tools 35

2.1. Standard Cryptographic Primitives

Let us briefly recall the main cryptographic primitives used in cryptographic protocols.

Symmetric encryption Symmetric cryptography refers to encryption methods in which both the sender and the receiver share the same key, such as the Advanced Encryption Standard (AES). Informally, a ciphertext $\mathrm{senc}(m, k)$ is computed by *encrypting* a message m with a key k, and the original message can be recovered by *decrypting* the ciphertext with the same key, *i.e.,* $\mathrm{sdec}(\mathrm{senc}(m, k), k) = m$. Intuitively, a symmetric encryption scheme is secure if an adversary cannot learn any information about the plaintext m from the ciphertext $\mathrm{senc}(m, k)$, without knowing the key k.

A significant disadvantage of symmetric ciphers is the key management necessary to use them securely. Each distinct pair of agents must share a different key. Therefore, the number of required keys increases as the square of the number of agents, which very quickly requires complex key management schemes to keep them all straight and secret. The difficulty of securely establishing a secret key between two agents, when a secure channel does not already exist between them, also presents a chicken-and-egg problem. This is why the recourse to asymmetric cryptography is so popular for key establishment protocols that aim to establish a fresh symmetric key between two parties.

Asymmetric encryption In 1976, Diffie and Hellman proposed the notion of public key cryptography, in which two different but mathematically related keys are used – a *public key* and a *private key*. A public key system is constructed, in such a way that calculation of the private key is computationally infeasible from the public key, even though they are necessarily related. In public key cryptosystems, the public key may be freely distributed, while its associated private key must remain secret. The public key is typically used for encryption, while the private key is used for decryption. Diffie and Hellman showed that public key cryptography was possible[1] by presenting the Diffie-Hellman key exchange protocol [DH76] in 1976. In 1978, Rivest, Shamir, and Adleman invented RSA, another public key cryptosystem [RSA78] which has established itself as the main standard.

Informally, a ciphertext for RSA is some $\mathrm{aenc}(m, \mathrm{pk}(sk))$ which is computed by *encrypting* a message m with a public key $\mathrm{pk}(sk)$, and the original message can be recovered by *decrypting* the ciphertext with the *associated*

1: Diffie and Hellman have been rewarded by the ACM Turing Award in 2016 for having laid the foundations of asymmetric encryption.

private key sk, *i.e.,* $\text{adec}(\text{aenc}(m, \text{pk}(sk)), sk) = m$. Intuitively, an asymmetric encryption scheme is secure if an adversary cannot learn any information about the plaintext m from the ciphertext $\text{aenc}(m, \text{pk}(sk))$, without knowing the private key sk.

Diffie-Hellman key exchange Diffie and Hellman proposed the Diffie-Hellman key exchange, a method for two parties to establish a shared secret key over an insecure channel. Classically, the parties work in a prime-order subgroup of $(\mathbb{Z}/p\mathbb{Z})^*$ generated by g, where p is a large prime. Alice and Bob uniformly sample $a, b \leftarrow \{1, \dots, p-1\}$ respectively. Alice sends $A = g^a \bmod p$ to Bob, and Bob sends $B = g^b \bmod p$ to Alice. They both derive the same shared secret $s = B^a \bmod p \equiv A^b \bmod p \equiv g^{ab} \bmod p$. Assuming the hardness of the Computational Diffie-Hellman (CDH) problem in this group, an attacker knowing A and B cannot compute s. We discuss a protocol based on Diffie-Hellman key exchange in Section 2.2.1.

Digital signature Over the same period, signature schemes have also been proposed. A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or of a document. It gives the recipient a reason to believe that the message was created by a known sender, that the sender cannot deny having sent the message (authentication and non-repudiation), and that the message was not altered while in transit (integrity).

Hash function A hash function takes a message of any length as input, and outputs a short, fixed length digest of this message. Hash functions have many security applications, notably in digital signatures, Message Authentication Code (MAC), and other forms of authentication. For good hash functions, an attacker cannot find two messages that produce the same hash —we revisit cryptographic assumptions for hash functions in Chapter 3. MACs are much like hash functions, except that a secret symmetric key can be used to authenticate the hash value upon receipt.

Authenticated Encryption with Associated Data (AEAD) Authenticated Encryption with Associated Data is a form of encryption that simultaneously provides confidentiality, integrity, and authenticity guarantees. Unlike plain symmetric encryption, AEAD schemes take as input a key, a plaintext, and additional associated data that is authenticated but not encrypted. This makes AEAD particularly suited to secure protocol design, where metadata such as headers or sequence numbers must be transmitted in the clear but still protected against tampering. Modern protocols such as TLS 1.3 and the Noise framework rely on AEAD as their core symmetric encryption primitive.

2.2. Security Protocols

In general, a cryptographic protocol is a distributed algorithm involving multiple communicating parties. It uses cryptographic primitives to achieve secure communication and other security goals, even in the presence of an adversary who may partially or fully control the network and compromise some participants. As we shall see throughout this manuscript, composing cryptographic primitives as building blocks into a secure protocol is a highly non-trivial and error-prone task. We present

in this section some examples of real-world protocols and their security properties.

2.2.1. Sigma', an Authenticated Key Exchange Protocol

Authenticated Key Exchange (AKE) protocols are cryptographic protocols that aim to establish a shared symmetric, session key between two parties and authenticate each other in the process. The two parties are then able to use the session key to securely exchange messages using symmetric encryption. Sigma' [BL16] is a simple AKE protocol, derived from the basic SIGMA-R (sign-and-mac) protocol from [Kra03], and very similar to other widely used AKE protocols such as the IKE and JFK protocols —both standardized by the Internet Engineering Task Force (IETF)— and the Off-the-Record Messaging protocol [BL16].

Sigma', depicted in Figure 2.1, involves an initiator (referred to as A) and a responder (referred to as B). The two first messages are essentially a Diffie-Hellman key exchange, where A and B exchange their Diffie-Hellman public values g^x and g^y. A and B add some protocol-specific parameters infoA and infoB to these messages, which may include, for example, a nonce, or some configuration options for negotiation. Both parties can then compute the shared Diffie-Hellman secret $s = g^{xy}$ and authenticate each other by signing the hash over the protocol transcript —the concatenation of the two first messages— using their long-term signing keys, and proving to each other that they know the session key s by exchanging MACs computed with this key over their own identities. Both parties then verify each other's signatures[2] and MACs before accepting the session key s as valid.

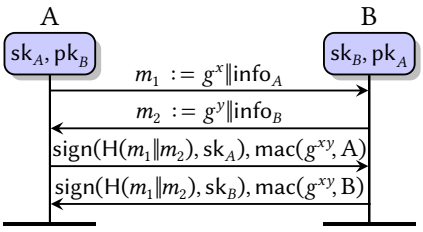


Figure 2.1.: The Sigma' protocol [BL16]. The operator $\cdot\|\cdot$ denotes concatenation.

2: We assume a Public Key Infrastructure (PKI) where parties know each other's public keys.

Such AKE protocols are designed to achieve a number of security properties, including:

- confidentiality of session keys: the adversary cannot learn the session key established by an honest party who engaged in a session with another honest party,
- mutual authentication: when an honest party believes it has completed a protocol session with another honest party, then this other party has a similar belief,
- mutual agreement: when an honest party believes it has completed a protocol session with another honest party, then this other party has a similar belief and they both agree on session information such as the session key, infoA, and infoB.

Crucially, defining these properties first require defining a precise threat model, *i.e.,* the adversary's capabilities and initial knowledge. As we shall see, a reasonable threat model to analyze such protocols is the DY adversary, which controls the network and can intercept, modify, and inject messages at will, but cannot break the underlying cryptographic primitives. Such protocols are also supposed to resist an insider attacker who can compromise some parties and their long-term keys. The sessions established between two honest parties should remain secure even if honest parties engage with compromised parties in *other* sessions. We revisit this protocol with some analysis results in Section 3.4.

2.2.2. The Noise Family of AKE Protocols [*Gir+20*]

The Noise Protocol Framework [Per18] specifies a family of two-party handshakes for establishing secure channels. In addition to specifying 59

handshakes and claiming various security properties for them, it also specifies how additional handshakes can be derived. The proposed uses are extremely broad, ranging from handshakes between unidentified parties to handshakes between parties having pre-shared static asymmetric and symmetric keys. One way to understand the framework is as a compact and simple language for describing a rich family of secure communication protocols.

Handshakes Each handshake specified by the Noise Protocol Framework is built from a succinct set of simple primitives: a Diffie-Hellman group, a hash function, a Key Derivation Function (KDF), and an AEAD cipher. Although the specification is written in a generic fashion, it limits the instantiation of said primitives to a small selection, with a rationale for each choice. Thus, the security properties ascribed to each handshake are only claimed to hold for the given instantiations.

Each handshake is described by a *pattern* following a simple grammar. A pattern has two parts: *pre-messages* and *messages*. Pre-messages describe setup assumptions, namely knowledge that the parties must share before starting the handshake, for example keys given by a PKI. Messages describe operations that each party must perform when sending or receiving handshake messages.

Pre-messages, messages, and computations thereon are described by a list of *tokens* and a *direction* specifying sender and recipient. Tokens refer to keys. Each party may have an ephemeral key (usually denoted by the letter `e`), and a static, or long-term, public key (usually denoted by the letter `s`). Additionally, the parties may share a secret called the Pre-Shared Key (PSK) (a symmetric key usually denoted as `psk`). Not all patterns require all these keys.

Definition 2.2.1 (Handshake pattern) *A* pre-message token *is* `e` *or* `s`. *A* message token *is* `e`, `s`, `es`, `se`, `ss`, `ee`, *or* `psk`. *Single letter tokens and* `psk` *are called* key tokens *and two-letter tokens are called* Diffie-Hellman tokens *(or DH tokens for short).*

A direction *is* → *or* ←. *A* pre-message *(respectively* message*) pattern is a pair of a direction and a non-empty list of pre-message (respectively message) tokens. A* handshake pattern *(or, for brevity, a handshake) has a name, and is a possibly empty list of pre-message patterns (followed by ellipsis if non-empty) and a non-empty list of message patterns. The list of pre-message patterns must contain at most one pre-message pattern per direction, and the message patterns must have alternating directions.* ◁

As an illustration, we depict three handshake patterns in Figure 2.2. Noise handshake participants always exchange an ephemeral public key with the other party, and may optionally also exchange a static (long-term) public key or one or more PSKs. The name of the handshake defines

NN:
→ e
← e, ee

KK:
→ s
← s
…
→ e, es, ss
← e, ee, se

I1K1:
← s
…
→ e, s
← e, ee, es
→ se

Figure 2.2.: Three Noise handshakes (colors match those of Figure 2.3, to help the reader, but are not part of the syntax)

Nomenclature	Informal meaning
N	No static key available
K	Static key known before (*e.g.,* via PKI)
X	Static key transmitted over the network
I	Static key transmitted earlier than with X
psk	Pre-shared symmetric key available
$n \in \mathbb{N}$	Appended to any other item, delays its use

Table 2.1.: Summary of Noise options and nomenclature for fundamental patterns. A fundamental pattern consists of two letters and an optional *psk* token. The letter I may only appear in the first position.

the *fundamental pattern* and is given by two letters indicating how static keys are used. There are four possibilities for the initiator, and three for the recipient: N means that no static key is available, K means the partner already knows the static key, and X means the static key is transmitted to the partner. For the initiator, I is also available and means the static key is transmitted immediately, improving authentication properties for the recipient, at the cost of revealing the initiator's identity and thus a loss of anonymity for the initiator. The handshake's first letter depends on the initiator's key and the second letter on the recipient's key. Table 2.1 contains a reference summary.

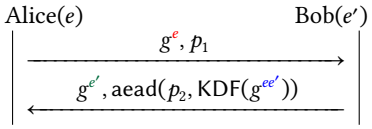
Note that the `psk` key token can be used in a message pattern to indicate that both parties should use their PSK and mix it with their current symmetric key. This adds *psk* to the handshake name, making it a non-fundamental pattern. Note that by default each available key computation and transmission is done as early as possible.

To delay certain actions, *deferred patterns* are created by adding a number to the pattern name, which defers the transmission of the named token (see I1K1 shown in Figures 2.2 and 2.3) or the use of derived keys, *e.g.,* `psk` (for *psk*), by a number of messages, usually 1 or 2. For example, *psk1* in a handshake name means that `psk` will be used one message after it could have been used. Deferred patterns were designed to improve identity hiding properties at the expense of latency.

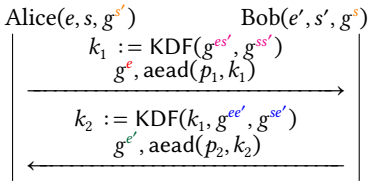
For the sake of brevity, we do not describe the formal semantics of tokens. We direct the interested reader to [Gir19]. In the next example, however, we try to give an intuition of the semantics of a few patterns, omitting some details for simplicity.

Example 2.2.1 (Handshake syntax and semantics) Consider *NN* (shown in Figures 2.2 and 2.3a), a Noise handshake loosely corresponding to an unauthenticated Diffie-Hellman key exchange. There are no pre-messages, so the ellipsis is omitted. In the first message, the initiator (on the left) sends their ephemeral public key g^e, indicated by the key token `e` in the first message pattern. Each message sent ends with a payload, protected using AEAD under the best available key (this step is not materialized as a token in the Noise syntax, but comes implicitly at the end of all message patterns). Here, p_1 is sent in the clear. In the second message, the recipient (on the right) sends their own ephemeral key $g^{e'}$, indicated by the key token `e` in the second message pattern. The token `ee` means that when processing the second message, both parties derive a Diffie-Hellman term from their respective ephemeral keys. Specifically, after the second message, the initiator knows their private key e and the recipient's public key $g^{e'}$. They can thus compute $g^{ee'} = (g^{e'})^e$. The recipient's situation is symmetrical: they know g^e and e' and can thus compute $g^{ee'} = (g^e)^{e'}$. Colloquially speaking, the DH term obtained by mixing the initiator's and recipient's ephemeral keys is $g^{ee'}$. They will use this value to seed a secret symmetric key, which is the initial *current key*, and is used to protect payloads, here p_2, with AEAD and the hash of all previous computation steps as additional data.

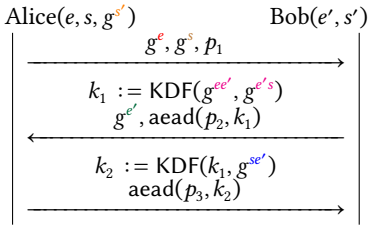
One can also mix different keys as DH terms. When several such terms are present, they can be *mixed* together using a KDF to obtain a new current key.[3] This is illustrated by KK, a Noise handshake loosely corresponding to an authenticated Diffie-Hellman key exchange shown in Figures 2.2 and 2.3b. Both pre-messages contain the same unique key token `s`, meaning both parties should already know their peer's public static key before starting the handshake. Namely, the initiator (respectively recipient) knows their private static key s (respectively s') and their



(a) NN handshake



(b) KK handshake



(c) I1K1 handshake

Figure 2.3.: Alice & Bob notation for the handshakes of Figure 2.2. e, e' (respectively s, s') are ephemeral (respectively static) private keys and the p_i are payloads exchanged during the handshake. In transport mode, payloads are encrypted with the last key material used in the handshake. For legibility, we omitted the associated data of AEAD encryptions, which roughly corresponds to the hash of all preceding sent messages along with the public keys in pre-messages.

3: We use *mix* in an overloaded manner, to both denote DH-computation with two half-keys, and KDF-application with two secrets.

partner's public static key $g^{s'}$ (respectively g^{s}). Then in the first message the initiator (i) sends g^{e} (key token `e`) in clear text because the current symmetric key is initially empty (*i.e.,* not yet determined), (ii) computes the DH term $g^{es'}$ (DH token `es`) and mixes it with the current symmetric key that is then no longer empty (*i.e.,* can now be used), (iii) computes the DH term $g^{ss'}$ (DH token `ss`) and mixes it with the current symmetric key (the resulting value is denoted as k_1 in Figure 2.3b), and finally (iv) sends the first payload protected using AEAD under the current symmetric key k_1 with the transcript of all messages exchanged so far as additional data. When receiving the corresponding message (*i.e.,* the pair $\langle g^{e}, c\rangle$, where c is the encrypted payload), the recipient performs the same computations and obtains the symmetric key k_1 and can therefore decrypt c.

For the second message, the recipient sends $g^{e'}$ (key token `e`), computes two DH terms corresponding to `ee` and `se`, and obtains the symmetric key k_2 accordingly. Similarly, the message ends with the second payload protected by AEAD with the key k_2 and the hash of all previous computation steps as additional data.

Finally, the transport mode can start where all payloads are protected with AEAD under a derivative of the final symmetric key k_2 and empty additional data. ◊

Examples of Real-World Uses The Noise Protocol framework has seen significant real-world adoption due to its flexibility and performance. A prominent example is the WireGuard VPN protocol [Don17], which uses the IK handshake to establish secure tunnels. WhatsApp [Inc17] employs Noise Pipes (a specific Noise construction) to secure communication between clients and its chat servers. The Lightning Network [Net17] for Bitcoin also uses Noise (specifically XK) for its authenticated transport layer (BOLT 8). Other notable implementations include Slack's Nebula overlay network, the I2P anonymity network, and various components of cloud infrastructures such as Amazon Web Services.

Security levels The specification [Per18] defines three source properties (degree of authentication of the sender provided to the recipient), six destination properties (degree of confidentiality provided to the sender), and ten identity-hiding levels (protection of the sender's or the recipient's public key), where higher numbers are supposed to indicate better security. The descriptions of these security properties are informal and non-trivial to interpret.

Example 2.2.2 Destination Property 4, quoted from [Per18]: "***Encryption to a known recipient, weak forward secrecy if the sender's private key has been compromised.*** *This payload is encrypted based on an ephemeral-ephemeral DH, and also based on an ephemeral-static DH involving the recipient's static key pair. However, the binding between the recipient's alleged ephemeral public and the recipient's static public key has only been verified based on DHs involving both those public keys and the sender's static private key. Thus, if the sender's static private key was previously compromised, the recipient's alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the intended recipient's static private key to decrypt the payload (this is a variant of a Key Compromise Impersonation (KCI) attack enabling a 'weak forward secrecy' attack).*"

This informal description discusses how the encryption key has been derived and describes a possible attack that an attacker could use. How-

ever, it does not explore any other circumstances in which the encryption key might be compromised or how this property relates to more traditional notions of message confidentiality and authentication. ◊

Example 2.2.3 Source Property 2, quoted from [Per18]: "***Sender authentication resistant to KCI.*** *The sender authentication is based on an ephemeral-static DH ('es' or 'se') between the sender's static key pair and the recipient's ephemeral key pair. Assuming the corresponding private keys are secure, this authentication cannot be forged.*"

The above definition is clearer than Example 2.2.2 insofar as it explicitly refers to a well-known and established definition. However, there is still considerable ambiguity. For example, is this authentication injective [Low97] (preventing replays)? ◊

We revisit Noise protocols in Chapter 5 when presenting our analysis results; we will show that our systematic approach allows us to clarify the above properties and to discover subtleties that are completely hidden or missing from the specification (see Section 5.5).

2.3. Dolev-Yao Model

Most of our protocol analyses are carried out in the so-called *Dolev-Yao (DY) model*[4], going back to the seminal work of Dolev and Yao [DY83]. This model is a formal abstraction of the real world.

4: Also called the *symbolic model*.

Central to the DY model is the assumption of a DY adversary, which has full control over the communication network and can intercept, modify, and inject messages at will. In a sense, the network is modeled as an adversary. However, it is assumed that the DY adversary cannot break the cryptographic primitives, that is they can only exploit the algebraic properties explicitly specified in the model.[5]

5: While this is often called the *perfect cryptography assumption* or *ideal cryptography assumption*, we prefer the term *black-box cryptography assumption* as it supports the idea that known cryptographic primitive weaknesses can be modeled by explicitely adding them to the model, and that the security of a protocol can be analyzed under different assumptions on the underlying primitives by changing the model accordingly. We discuss this in more detail in Chapter 3.

These central assumptions and abstractions allow modeling cryptographic messages as formal terms in a term algebra equipped with an equational theory, and cryptographic primitives as idealized black boxes that only satisfy the properties explicitly specified by algebraic equations. Cryptographic protocols are modeled in formal languages such as the applied pi-calculus [AF01] or multiset rewriting systems [Sch+12a; Sch12; Mei13], and security properties are expressed as reachability or behavioral equivalence properties over these models. This level of abstraction places the focus on the logical structure of protocols and their interactions, and allows for the use of powerful automated verification tools to analyze them to find so-called *logical attacks* that exploit protocol logic flaws.

We present in the next sections the main concepts and techniques of the DY model, as well as the main tools for automated verification in this model. We focus on the most mature and widely used ones (Tamarin and ProVerif), and briefly discuss other tools in Section 2.4.3.

2.3.1. Cryptographic Messages Models

In the DY model, messages are described by terms from a term algebra. For example, $\mathrm{senc}(m, k)$ represents the message m encrypted using the key k. The algebraic properties of cryptographic functions are specified by equations over terms. For example, $\mathrm{sdec}(\mathrm{senc}(m, k), k) = m$ specifies the expected semantics for symmetric encryption: decryption using the encryption key yields the plaintext. As is common in the symbolic model,

cryptographic messages only satisfy those properties explicitly specified algebraically. This yields the now standard *black-box cryptography assumption*: one cannot exploit potential weaknesses in cryptographic primitives beyond those explicitly specified. We detail next the message model used by Tamarin, though the main concepts are similar in other tools.

Tamarin message model Formally, we represent messages and operations on them as *terms* in an *order-sorted term algebra*. For this, we assume a set of *operators* with their arities as *signature* Σ. We define three sorts, a top sort *msg* including all terms together with two incomparable subsorts *fr* and *pub*, where terms of the former model random ("fresh") values in general, and nonces and keys in particular, while terms of the latter represent publicly known ("public") values. We also assume countably infinite sets of *variables* $\mathscr{V}_s$, for each sort s, and call the union of all such sets $\mathscr{V}$. We similarly treat *names*, with a countable set $\mathscr{A}_s$ for each sort s, and their union $\mathscr{A}$. The set of *terms* given by the closure of using operators from the signature containing variables in $\mathscr{V}$ and names in $\mathscr{A}$ is denoted $T_\Sigma(\mathscr{V}, \mathscr{A})$. A term t is *ground* if it contains no variable, and we write $T_\Sigma(\mathscr{A})$ for the set of all ground terms, or simply T_Σ. A *substitution* σ is a function from variables to terms. We homomorphically lift substitutions to terms and use postfix notation, so $\sigma(t)$ is written $t\sigma$.

Given a signature Σ, an *equation* is an unordered pair of terms s and t, written $s = t$, for $s, t \in T_\Sigma(\mathscr{V}, \emptyset)$. The *equational theory* associated to a set of equations E and a signature Σ is the smallest congruence closure containing all instances of E, for which we write $=_E$ when Σ is clear from the context. We consider sets, sequences, and multisets modulo E by using the subscript E (*e.g.*, $\in_E$).

Example 2.3.1 Diffie-Hellman exponentiation can be modeled by an operator $\exp(\cdot,\cdot) \in \Sigma$ together with the equations[6] $\exp(x, 1) = x$ and $\exp(\exp(x, y), z) = \exp(x, \exp(y, z))$. ◊

6: Tamarin is actually able to handle an extension of this theory modeling the abelian group for the exponents.

Example 2.3.2 One may also consider a pairing and projections operators $\langle\cdot,\cdot\rangle, \pi_1(\cdot), \pi_2(\cdot) \in \Sigma$ equipped with the equations $\pi_i(\langle x_1, x_2\rangle) = x_i$ for $i \in \{1, 2\}$. ◊

Example 2.3.3 (Term) Assuming concatenation is modeled by pairing and projections as in Example 2.3.2, the first message of the Sigma' protocol from Figure 2.1 can be represented by the term $\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle$ where g is a constant —an operator of arity 0— modeling a public generator, and $n_x, \mathsf{info}_\mathsf{A} \in \mathscr{A}$ are names. ◊

Example 2.3.4 To model asymmetric signatures, one can introduce the operators $sign(\cdot,\cdot)$, $checksign(\cdot,\cdot)$, and $pk(\cdot)$ together with the equation $checksign(sign(x, k), pk(k)) = x$. ◊

As we shall see, not all equational theories will be supported by the verification tools we use.[7] For instance, Tamarin supports a number of built-in theories such Diffie-Helman exponentiation, bilinear pairing, multisets, *etc.* One can also define custom theories by providing a set of equations provided that the theory should be convergent and have the FVP [CD05] modulo AC, see Definition 2.4.2. This captures a wide range of theories, including the ones used in this manuscript, except when we extended the theory and the tool ourselves to extend its scope and expressivity —as we did for XOR (Section 4.1.1) and associative pairing (Section 4.1.2).

7: ProVerif supports a combination of linear and subterm-convergent theories, while Tamarin supports a wider class of theories, including all convergent theories with the Finite Variant Property (FVP) property modulo Associative-Commutative (AC).

2.3.2. Protocol Models

Two main formalisms have been developed to model security protocols in the symbolic model: the applied pi-calculus [AF01] (notably used by

ProVerif, DeepSec, and Sapic+) and Multiset Rewriting Rule (MSR) [Sch+12a; Sch12; Mei13] (notably used by Tamarin). While the two formalisms are quite different, they are both expressive enough to model a wide range of protocols and security properties. We present next the MSR formalism used by Tamarin and we refer the reader to [AF01] for the applied pi-calculus.

Tamarin Protocol Model: Multiset Rewriting Rule (MSR) [Sch+12a; Sch12; Mei13] Cryptographic protocols can be modeled by a set of *MSRs*. These rules operate on multisets of *facts* containing terms, which represent the current state of the system and the adversary's knowledge.[8] Intuitively, a state shall be modeled as a multiset of facts and MSRs can be applied to transform one state into another by consumption and creation of facts.

Formally, we assume an unsorted *fact signature* Σ_{Fact}, partitioned into *linear and persistent facts*. The set of *facts* is defined as $\mathscr{F} = \{F(t_1, \dots, t_n) \mid t_i \in T_\Sigma(\mathscr{V}, \mathscr{A}), F \in \Sigma_{Fact} \text{ of arity } n\}$. Linear facts can only be consumed as often as they have been created, while persistent facts can be consumed an unbounded number of times. We denote the set of multisets of facts as $\mathscr{F}^\#$, and the set of multisets of grounded facts as $\mathscr{G}^\#$. The function set(·) converts multisets to sets by dropping the multiplicity. We use the superscript $^\#$ to denote operations on multisets, *e.g.,* $\cup^\#$ denotes the union on multisets. For $l \in \mathscr{F}^\#$, we denote by $lfacts(l)$ the sub-multiset of all its linear facts and by $pfacts(l)$ the sub-multiset of all its persistent facts. A *state* is a multiset of grounded facts, *i.e.,* an element of $\mathscr{G}^\#$.

A protocol shall be modeled by a set of *labeled MSRs*, which describe the possible transitions between states. Such rules are given as (id, l, a, r), and are written as $id : l \text{–}[\, a \,]\!\to r$, with id a unique identifier, and l, a, and r multisets of facts. We say its *name* is $name(ri) = id$, its *premises* which will be consumed from the sate are $prems(ri) = l$, its *conclusions* which will be produced and added to the state are $concs(ri) = r$, and its *actions* which will be attached to the transition are $acts(ri) = a$.

The set of ground instances of a set of MSRs R is denoted $ginsts(R)$. The semantics of a set of MSRs R are given by a *labeled transition relation* $\to_R \subseteq \mathscr{G}^\# \times \mathscr{G}^\# \times \mathscr{G}^\#$, defined by the following *step rule*, where $S \in \mathscr{G}^\#$ is the current state, *i.e.,* a multiset of grounded facts:

$$\frac{ri = id : l \text{–}[\, a \,]\!\to r \in_E ginsts(R) \quad lfacts(l) \subseteq^\# S \quad pfacts(l) \subseteq S}{S \xrightarrow{\text{set}(a)}_R ((S \setminus^\# lfacts(l)) \cup^\# r)}$$

Note that the initial state of a labeled transition system derived from MSRs is the empty multiset of facts $\emptyset$. The transition according to the step rule transforms a multiset of facts into another multiset of facts. Linear facts are consumed and need to be available sufficiently often, while persistent facts just need to be present. The next state is derived by removing consumed linear facts and adding all generated linear (with correct multiplicity) and persistent facts. Note that we rewrite modulo an equational theory E, so we use $\in_E$ for the rule instance modulo. Finally, the *actions* of the rule (set(a)) are the label attached to the transition. We shall model the security properties by logical formulas over these labels.

There is a single distinguished *fresh* rule: $Fresh : \text{–}[\,]\!\to \mathsf{Fr}(n)$. This rule has no premise and is the only rule allowed to create the linear Fr facts. To ensure that the generated n is unique, there is an additional condition enforcing that within an execution the variables n from two instances are

8: For example, the fact operator $\mathsf{A}_1(\cdot,\cdot,\cdot)$ could represent the state of an initiator after sending the first message of the Sigma' protocol, *e.g.,* $\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A})$. The fact $\mathsf{K}(\cdot)$ represents the adversary's knowledge, *e.g.,* $\mathsf{K}(\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle)$ would represent that the adversary has learned the first message of the Sigma' protocol.

Example 2.3.5 For instance, the rule R_1 modeling the sending of the first message of the Sigma' protocol is:

$$\mathsf{R}_1 : \{\mathsf{Fr}(n_x), \mathsf{Fr}(\mathsf{info}_\mathsf{A})\} \text{–}[\,\mathsf{Start}\,]\!\to \{\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A}), \mathsf{Out}(\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle)\}$$

where the linear facts $\mathsf{Fr}(\cdot)$ models the generation of fresh values, $\mathsf{Out}(\cdot)$ models the sending of messages, A_1 models the initiator's state after sending the first message, and n_x, info_A have sort *fr*.
The same rule can be written as:

$$\frac{\mathsf{Fr}(n_x),\ \mathsf{Fr}(\mathsf{info}_\mathsf{A})}{\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A}),\ \mathsf{Out}(\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle)}[\mathsf{Start}]$$

The simplified protocol model $P_{Sigma'} = \{R_1, R_2\}$ has an additional rule R_2:

$$\frac{\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A}),\ \mathsf{In}(\langle y_{\mathsf{exp}}, y_{\mathsf{info}}\rangle)}{\mathsf{A}_2(n_x, \mathsf{info}_\mathsf{A}, y_{\mathsf{exp}}, y_{\mathsf{info}}, \exp(y_{\mathsf{exp}}, n_x))}[a],$$

where $a \triangleq \mathsf{SessKey}_\mathsf{A}(\exp(y_{\mathsf{exp}}, n_x))$, y_{exp} and y_{info} have sort *msg* (any message), and n_x and info_A have sort *fr* (fresh values). This rule models the initiator receiving the second message (linear fact $\mathsf{In}(\cdot)$) and storing it in state fact A_2 along with the session key $\exp(y_{\mathsf{exp}}, n_x)$, which models g^{xy} from Figure 2.1. The session key is recorded in the action fact $\mathsf{SessKey}_\mathsf{A}$, used to express security properties. ◊

different. A detailed explanation is available in [Sch+12a].

We define an *execution* e of a protocol P, modeled as a set of MSRs, as the alternating sequence of states and rule instances: $S_0, (l_1 -[\ a_1\ ]\!\!\rightarrow r_1), S_1, \ldots, S_{n-1}, (l_n -[\ a_n\ ]\!\!\rightarrow r_n), S_n$ such that $S_0 = \emptyset$, and that for all $i \in \{1, \ldots, n\}$ we have all $(S_{i-1}, (l_i -[\ a_i\ ]\!\!\rightarrow r_i), S_i)$ are valid steps according to the step rule. We associate the *trace* $\text{trace}(e) = [set(a_1), \ldots, set(a_n)]$ with such an execution e. We write $\text{exec}(P)$ for the set of executions of P.

Recall that we consider a DY adversary with full control over the network and the ability to apply all operators. The message deduction rules are given by MD depicted in Equation (2.1). The adversary learns all messages sent by participants as they are put into the linear $\mathsf{Out}(\cdot)$ facts and are subsequently stored in the persistent adversary knowledge $\mathsf{K}(\cdot)$. The adversary can send messages to protocol participants by putting them into linear $\mathsf{In}(\cdot)$ facts. Moreover, the adversary can generate their own random values and knows all public values. They can also apply functions from the signature using the rules in the second line of MD.

Example 2.3.6 A sample execution e of a protocol with at least the aforementioned rules is given by the following sequence of states and rule instances:
$S_0 = \emptyset$, $(Fresh : -[]\!\!\rightarrow \mathsf{Fr}(n_x))$,
$S_1 = \{\mathsf{Fr}(n_x)\}$, $(Fresh : -[]\!\!\rightarrow \mathsf{Fr}(\mathsf{info}_\mathsf{A}))$,
$S_2 = \{\mathsf{Fr}(n_x), \mathsf{Fr}(\mathsf{info}_\mathsf{A})\}$,
$(\mathsf{A}_1 : \{\mathsf{Fr}(n_x), \mathsf{Fr}(\mathsf{info}_\mathsf{A})\} -[\ \mathsf{Start}\ ]\!\!\rightarrow \{\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A}), \mathsf{Out}(\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle)\})$,
$S_3 = \{\mathsf{A}_1(n_x, \mathsf{info}_\mathsf{A}), \mathsf{Out}(\langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle)\}$. The trace is $trace(e) = [\mathsf{Start}]$. ◊

$$\begin{aligned} MD = \{\ & \mathsf{Out}(x) -[]\!\!\rightarrow \mathsf{K}(x),\ \mathsf{K}(x) -[\ \mathsf{K}(x)\ ]\!\!\rightarrow \mathsf{In}(x),\ \mathsf{Fr}(x : fr) -[]\!\!\rightarrow \mathsf{K}(x : fr),\ [] -[]\!\!\rightarrow \mathsf{K}(x : pub)\ \} \\ & \cup \{\ \mathsf{K}(x_1), \ldots, \mathsf{K}(x_n) -[]\!\!\rightarrow \mathsf{K}(f(x_1, \ldots, x_n)) \mid f \in \Sigma \text{ with arity } n\ \} \end{aligned} \tag{2.1}$$

2.3.3. Security Property Models

Most security properties can be modeled as *reachability properties*, also called *trace properties* —a bad state cannot be reached— or *observational equivalence properties*, also called *indistinguishability properties* —two systems cannot be distinguished. The former are generally expressed as first-order logic formulas over execution traces, *e.g.,* there is no reachable state in which the adversary knows a secret[9]. The latter are expressed as *bisimulation* or *trace equivalence* of two systems, *e.g.,* the adversary cannot distinguish a system where a user is anonymous from a system where the user is not anonymous. Anonymity, unlinkability, and more generally privacy properties are typically of this form.

9: Confidentiality, authentication, and agreement properties are typically of this form.

Tamarin Formalism for Reachability Properties In Tamarin, but also in ProVerif, reachability properties are expressed as first-order logic formulas. Formulas use variables of a new sort, *temp*, for reasoning about the order of events, and are evaluated on traces. The informal semantics of the atomic formulas are

- $\bot$: false;
- $t_1 \approx t_2$: t_1 and t_2 are equal in the equational theory;
- $F@i$: fact $F \in_E tr[i]$ where i is of sort *temp* and $tr[i]$ is the i-th element of the trace tr on which we evaluate the formula;
- $i \doteq j$: timepoints i and j are equal;
- $i \lessdot j$: timepoint i occurs before timepoint j.

See [Sch+12a] for all the details on the semantics and fragment of first order logic accepted by Tamarin.

Example 2.3.7 (Continuing Example 2.4.1) Assuming we extend the protocol model $P_{Sigma'}$ with rules modeling the rest of the protocol, one could attempt to model the secrecy of the session key $\exp(y_{\text{exp}}, n_x)$ from the intiator point of view (A) with the following formula:

$$\forall x, i, j.\ \mathsf{SessKey}_\mathsf{A}(x)@i \wedge \mathsf{K}(x)@j \implies \bot$$

Example 2.3.8 (Continuing Example 2.3.7) Authentication properties rely on other action facts. For example, to model the fact that the initiator correctly authenticates the responder at the end of a Sigma' session, we could use the following formula:

$$\forall \mathsf{pk}_A, \mathsf{pk}_B, k, i.\ \mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)@i \implies \exists j.\ \mathsf{Start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)@j$$

where $\mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ logs that the initiator (identified by pk_A) has received and validated the last responder's message (where the responder is identified by pk_B) in a session that established the session key k and $\mathsf{Start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ logs that the responder started a session. ◊

When asked to verify this property, Tamarin will instantly return an attack trace. Indeed, the adversary only has to send their own adversary-generated Diffie-Hellman share $\exp(g, a)$ to the initiator, who will then compute the session key $\exp(\exp(g, a), n_x)$ that the adversary will also be able to compute from the initiator's public Diffie-Hellman share $\exp(g, n_x)$ and their own private key a.
This is actually a modeling error: the Sigma' protocol provides the confidentiality of the session key only for keys generated in sessions that are complete. That is, key confidentiality is not guaranteed prior to both peers sending and checking each other's signatures and MACs. To reflect this in the model, we would need to use the action fact $\mathsf{SessKey}_\mathsf{A}$ (and corresponding $\mathsf{SessKey}_\mathsf{B}$ for the responder) only in the last rules of the initiator and responder. Doing so, Tamarin would then be able to verify the secrecy of the session key. ◊

Note that some properties can be declared as *axioms* (called *restrictions* in Tamarin) instead of being proved as theorems: they are not proved but asserted, and are used to rule out any execution that violates them. They are typically useful for modeling assumptions on the protocol or primitives.[10]

Tamarin Formalism for Equivalence Properties Equivalence properties are modeled by the means of *diff-terms* (defining bi-systems, *i.e.,* two systems differing only in some terms) and check their observational equivalence, see [BDS15].

Example 2.3.10 ([BDS15], Ex. 10) An equational theory representing probabilistic encryption is

$$pdec(penc(m, pk(k), r), k) = m$$

This equation gives rise to the *decryption rule* for probabilistic encryption for the adversary automatically generated by Tamarin:

$$Dpenc \;:\; \mathsf{K}(penc(m, pk(k), r)), \mathsf{K}(k) \text{–}[\,]\!\to \mathsf{K}(m)\,.$$

Consider now the following bi-system:

$$\begin{aligned} S = \{\quad G :\quad & \mathsf{Fr}(k) \text{–}[\,]\!\to \mathsf{Key}(k), \mathsf{Out}(pk(k)), \\ E :\quad & \mathsf{Key}(k), \mathsf{Fr}(r_1), \mathsf{Fr}(r_2), \mathsf{In}(x) \text{–}[\,]\!\to \mathsf{Out}(diff[r_1, penc(x, pk(k), r_2)])\} \end{aligned}$$

Here Tamarin will compare the system where $diff[r_1, penc(x, pk(k), r_2)]$ is replaced by r_1 to the system where it is replaced by $penc(x, pk(k), r_2)$. If the adversary cannot distinguish the two systems, they are said to be observationally equivalent. In this example, this means that they cannot distinguish a probabilistic encryption from random. ◊

Example 2.3.9 (Noise security properties (Section 2.2.2)) First-order logic formulas are expressive enough to model a wide range of security properties, including complex authentication and agreement properties with various threat scenarios —*e.g.,* resistance to some entities' or data compromise and adversarial capabilities. For instance, all the security claims of the Noise protocol framework [*Gir+20*] can be modeled in such a formalism, including:

- *Secrecy* of a particular payload at position $i \in \mathbb{N}$, from the perspective of a given role r.
- *Non-injective agreement* after accepting a payload at position $i \in \mathbb{N}$ and on the payload contents and any non-empty set of (supposedly) exchanged keys from the perspective of the payload's recipient: the payload was at some point sent by a party with matching keys. However, there is no particular correspondence between these events, *e.g.,* it may be that the payload was only sent once, but received multiple times.
- *Injective agreement* additionally requires that any successfully received message must correspond to a legitimate transmission, ruling out replay attacks.

◊

10: For instance, for modeling a protocol agent checking that two terms are different in Tamarin, one can use an action fact $\mathsf{Neq}(\cdot,\cdot)$ in the corresponding MSR. One then declares a restriction $\forall x, y, i.\ \mathsf{Neq}(x, y)@i \Rightarrow x \neq y$ to rule out any graph where two terms that are supposed to be different are actually equal.

2.4. Automated Verification

Modelling security protocols in the symbolic approach enables machine support and, in many cases, fully automated push-button verification. However, the underlying verification problem is undecidable in general [Hüt03; CCD15]. Two main approaches address this. The first is to recover decidability by restricting the analysis to a bounded number of sessions[11] (and thus a bounded number of agents). The second is to semi-decide the problem for an unbounded number of sessions: soundness is guaranteed —if the tool returns a proof, the property holds— but

11: We return to this line of work in Section 4.2.1.

completeness is not, as the tool may fail to terminate. In practice, these tools offer interactive and heuristic-guided proof search, and termination is routinely achieved, allowing large classes of protocols and properties to be verified in the unbounded setting.

Most of our results are obtained using tools in this family, of which the more advanced ones are Tamarin and ProVerif. We first present the Tamarin prover and its underlying algorithm (Section 2.4.1), and then we briefly present the ProVerif prover (Section 2.4.2). We also mention other techniques and tools developed for the bounded number of sessions case (Section 2.4.3).

While these tools and their theoretical underpinnings are central to our work, not all the details are necessarily required to understand all of our results. Accordingly, the reader may skim the remainder of this section on a first reading and refer back to it as needed.

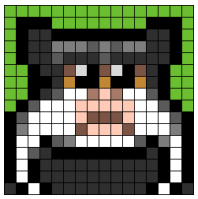


Figure 2.4.: The Tamarin prover logo.



Figure 2.5.: The ProVerif prover logo.

2.4.1. The Tamarin Prover

Tamarin[12] is an automated protocol verification tool in the DY model that semi-decides[13] the verification problem for an unbounded number of sessions. In fully automatic mode, Tamarin either returns a proof that the property holds, or a counterexample —representing an attack— if the property is violated, or it may fail to terminate as the underlying problem is undecidable. Tamarin can also be used in interactive mode, where users can guide the proof search. Moreover, users can supply heuristics to guide the proof search in a sound way.

12: Tamarin was originally designed and developed in the Simon Meier's and Benedikt Schmidt's PhD thesis [Mei13; Sch12] and has been continuously improved since then.

13: It is proven sound and complete but termination is not guaranteed, which is unavoidable given the undecidability of the underlying problem.

Tamarin analyzes the reachability of attack states, which are defined by a given security property. It does so by exploring symbolic traces in a backward fashion through a dedicated constraint solving procedure. We now give an overview of the underlying algorithm underpinning Tamarin, and the main techniques it relies on to achieve automation. The overall workflow is depicted in Figure 2.6.

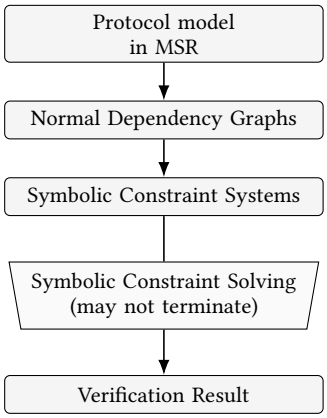


Figure 2.6.: Tamarin's verification workflow. All steps (→) are sound & complete. The constraint solving procedure also takes the negation of the security property to verify as input.

2.4.1.1. Dependency Graphs

To tame the complexity of reasoning about executions, Tamarin uses a more abstract representation of executions called *dependency graphs* which symbolically represent large classes of executions and on which symbolic reasoning will be carried out by the means of symbolic constraint solving.

Definition 2.4.1 (Dependency Graph [Sch+12a]) *We say that the pair* $dg = (I, D)$ *is a* dependency graph *modulo* E *for set of rules* R *if* I *is a sequence of ground instances (modulo* E*) of rules from* $R \cup Fresh$*,* $D \in \mathbb{N}^2 \times \mathbb{N}^2$*, and* dg *satisfies the conditions:*

DG1 *For every edge* $(i,u) \rightarrowtail (j,v) \in D$ *it holds that* $i \leq j$ *and the conclusion fact of* (i,u) *is equal to the premise fact of* (j,v)*.*
DG2 *Every premise of* dg *has exactly one incoming edge.*
DG3 *Every linear conclusion of* dg *has at most one outgoing edge.*
DG4 *The Fresh instances are unique.*

We denote the set of dependency graphs modulo E *for* R *by* $dgraphs_E(R)$*.* ◁

A dependency graph dg naturally gives rise to a set of executions by linearizing the graph dependencies (D), that we note exec(dg). The set of traces thereof is denoted trace(dg) It is easy to see that the dependency

Example 2.4.1 Continuing Example 2.3.6, Figure 2.7 gives a sample execution of the protocol $P_{Sigma'} \cup MD$, as a dependency graph. It also illustrates how the dependency graph represents the trace and intermediate states. ◊

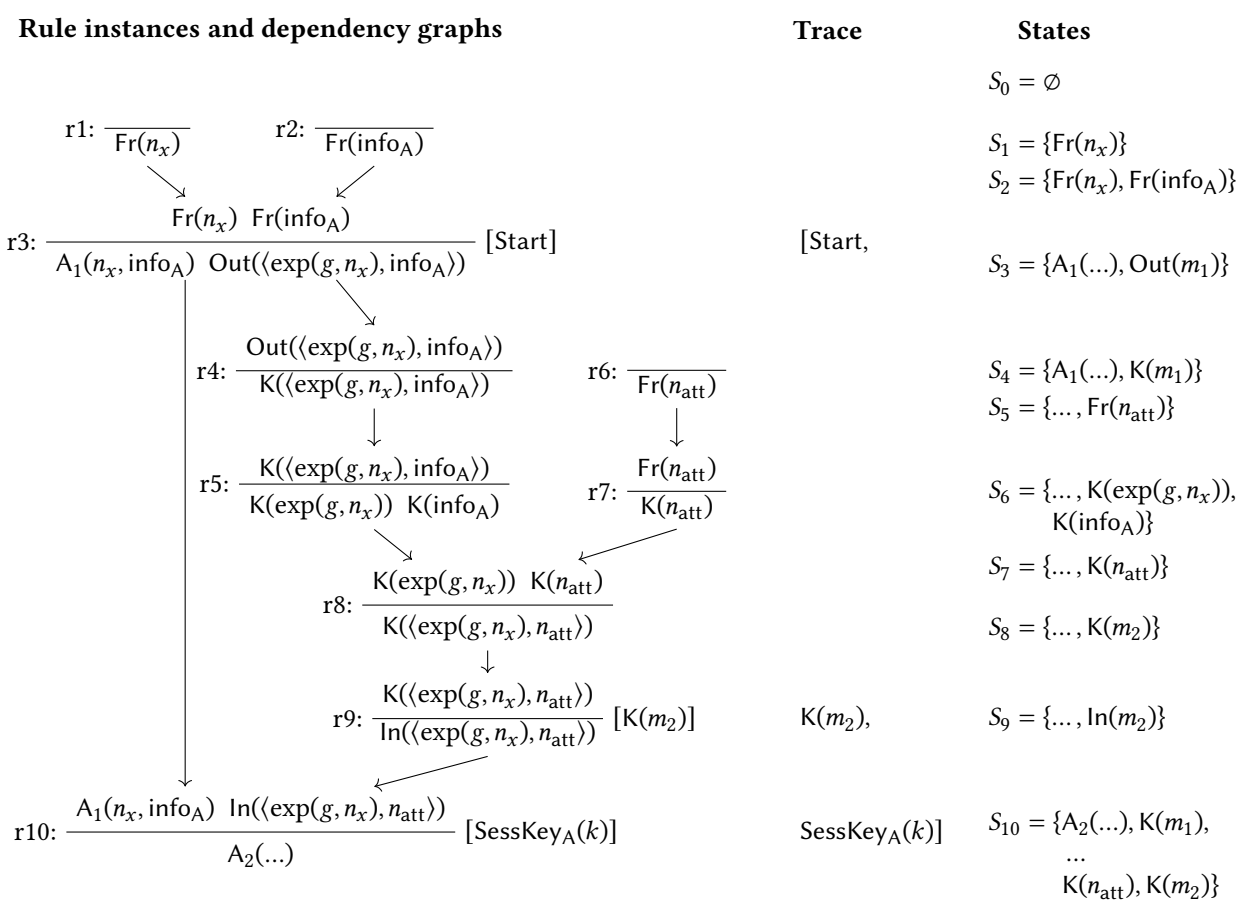


Figure 2.7.: Example execution of $P_{Sigma'} \cup MD$, where $m_1 \triangleq \langle \exp(g, n_x), \mathsf{info}_\mathsf{A}\rangle$, $m_2 \triangleq \langle \exp(g, n_x), n_{\mathsf{att}}\rangle$, and $k \triangleq \exp(\exp(g, n_x), n_x)$. The arrows denote causal dependencies. In particular, these dependencies correspond to the edges D of the underlying dependency graph, where $I \triangleq [\mathrm{r1}, \ldots, \mathrm{r10}]$. In this execution, the adversary intercepts the first initiator's message and replays the Diffie-Hellman share along with the adversarially-chosen nonce n_{att} (instead of info_B). Rules r5 and r8 are instances of the projector π_1 (deconstruction) and the pairing operator $\langle \cdot, \cdot \rangle$ (construction), respectively; r7 is the adversary-fresh rule that allows the adversary to generate fresh values. The trace of this execution is $[\mathsf{Start}, \mathsf{K}(m_2), \mathsf{SessKey}_\mathsf{A}(k)]$.

graphs have the same traces as the labeled transition system executions (Lemma 2.4.1)

Lemma 2.4.1 ([Sch12], Lemma 3.10) *Let E be an equational theory. Then, for all protocols P (i.e., a set of MSRs), $\{t \mid t \in \mathrm{trace}(e), e \in \mathrm{exec}(P \cup MD)\} =_E \{t \mid t \in \mathrm{trace}(dg), dg \in dgraphs_E(P \cup MD)\}$.*

2.4.1.2. Normal Dependency Graphs

To automate analysis, Tamarin restricts dependency graphs to *normal dependency graphs*. First, it handles equational theories via term variants and requires terms to be in normal form. Second, it imposes structural restrictions to prevent redundant and infinite explorations. Yet, these restrictions will guarantee that all relevant executions are represented by normal dependency graphs.

Term Normalization To enable efficient symbolic analysis modulo E, Tamarin splits E into a convergent rewriting part E' and the AC axioms AC, so that $E = E' \cup AC$. Since E' is assumed convergent modulo AC (see Section 2.3.1), every term t has a unique normal form modulo AC, written $t{\downarrow_E}$, satisfying $t =_E t{\downarrow_E}$ and $t{\downarrow_E}$ is irreducible by E'.

To avoid computing normal forms modulo AC during verification, Tamarin pre-computes *term variants*. As mentioned in Section 2.3.1, user-defined theories must be AC-convergent and satisfy the FVP [CD05],

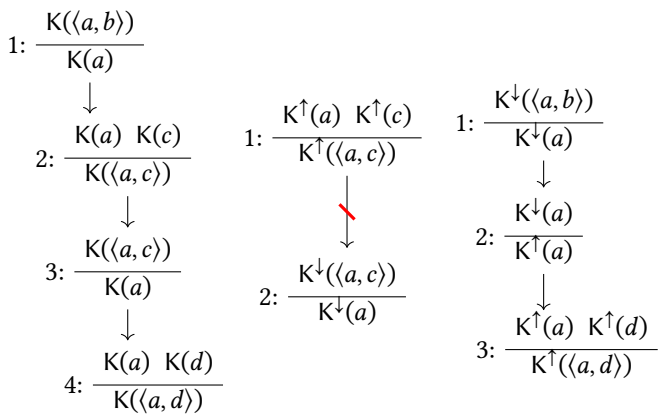


Figure 2.8.: Message deduction graphs for pairing. On the left is a redundant dependency graph with normalized terms. In the middle is a dependency graph violating the normal form conditions (forbidded ordering of K(·)-facts). On the right is a shorter deduction with final conclusion equivalent to the left, which is in normal form.

which we now define.

Definition 2.4.2 (FVP [CD05], simplified) *Given the decomposition $E = E' \cup AC$, a* complete set of variants of *t is a set S of terms such that, for every substitution σ, there is a term $t' \in S$ and a substitution θ such that $t\sigma{\downarrow_E} =_{AC} t'\theta$. We denote by $\lceil t \rceil^E$ one such complete set. The pair (E', AC) has the* FVP *if such sets can be effectively computed for all terms.* ◁

In practice, $\lceil t \rceil^E$ is computed via *folding variant narrowing* [ESM12]. The notation extends naturally to MSRs and protocols: for a rule r, $\lceil r \rceil^E$ denotes the set of its variants (obtained by computing variants of all terms in the rule), and for a protocol P, $\lceil P \rceil^E = \bigcup_{r \in P} \lceil r \rceil^E$.

By pre-computing $\lceil P \rceil^E$, Tamarin replaces reasoning modulo E with reasoning modulo AC only, for which unification can be efficiently decided (Tamarin delegates AC-unification to Maude [Cla+07]). Lemma 2.4.2 states that this pre-computation is sound and complete: the dependency graphs of the original protocol P modulo E coincide (up to normalization) with those of the variant-expanded protocol $\lceil P \cup MD \rceil^E$ modulo AC.

Lemma 2.4.2 ([Sch+12b] (extended version of [Sch+12a]), Lemma 5) *For all protocols P, $dgraphs_E(P \cup MD)\downarrow_E =_{AC} \{dg \mid dg \in dgraphs_{AC}(\lceil P \cup MD \rceil^E) \wedge dg \downarrow_E\text{-normal}\}$.*

Normalized Dependency Graphs Even for simple convergent theories E' containing only the pairing function $\langle \cdot, \cdot \rangle$ and operators $\pi_1(\cdot)$ and $\pi_2(\cdot)$, dependency graphs with normalized terms do not suffice to automate trace analysis. For example, the adversary may deduce a from $\langle a, b \rangle$ via $\pi_1(\cdot)$, pair it with c, extract a again from the new pair, and build $\langle a, d \rangle$. This is visualized in the left-most graph of Figure 2.8.[14] This dependency graph is correct but entirely redundant: the steps involving c could be skipped. Such redundancy is not only inefficient but also causes nontermination, as the adversary could insert arbitrarily many useless steps. To prevent this without compromising soundness –keeping all relevant traces–, Tamarin uses *normal* dependency graphs —dependency graphs satisfying *normal form conditions*— that exclude such useless steps. We first introduce the notion of construction and deconstruction rules needed to define these conditions.

14: Note that the topmost rule is actually an instance of the function application rule for $\pi_1(\cdot)$ where the conclusion $\pi_1(\langle a, b \rangle)$ is presented in the reduced form a, according to the equational theory.

Construction and Deconstruction Rules To avoid this redundancy, Tamarin makes the equational theory explicit by splitting the adversary rules (from MD) into *construction rules* and *deconstruction rules* (an example is shown in Figure 2.9). Deconstruction rules apply operators that reduce according to the equational theory, such as $\pi_1(\cdot)$, $\pi_2(\cdot)$, or decryption. They are typically used right after protocol rules that output messages (with Out(·)) to deduce further messages. Construction rules,

$$\left\{ \frac{\mathsf{Out}(x)}{\mathsf{K}^{\downarrow}(x)} \quad \frac{\mathsf{K}^{\uparrow}(x)}{\mathsf{In}(x)}[\mathsf{K}(x)] \quad \text{Coerce} : \frac{\mathsf{K}^{\downarrow}(x)}{\mathsf{K}^{\uparrow}(x)} \quad \frac{\mathsf{Fr}(x : fr)}{\mathsf{K}^{\uparrow}(x : fr)} \quad \frac{}{\mathsf{K}^{\uparrow}(x : pub)} \right\}$$

Construction rules: $\dfrac{\mathsf{K}^{\uparrow}(x_1) \;\; \mathsf{K}^{\uparrow}(x_2)}{\mathsf{K}^{\uparrow}(\langle x_1, x_2 \rangle)}$ Deconstruction rules: $\dfrac{\mathsf{K}^{\downarrow}(\langle x, y \rangle)}{\mathsf{K}^{\downarrow}(x)}$ $\dfrac{\mathsf{K}^{\downarrow}(\langle x, y \rangle)}{\mathsf{K}^{\downarrow}(y)}$

Figure 2.9.: Set of normal deduction rules ND when $\Sigma = \{\langle\cdot,\cdot\rangle, \pi_1(\cdot), \pi_2(\cdot)\}$.

conversely, build messages from the adversary's knowledge to be sent to the protocol (with $\mathsf{In}(\cdot)$). To enforce this, adversary knowledge K facts carry an *orientation*, *up* or *down*, denoted $\mathsf{K}^{\uparrow}$ and $\mathsf{K}^{\downarrow}$. Deconstruction rules have premises with both $\mathsf{K}^{\downarrow}$ and $\mathsf{K}^{\uparrow}$ facts (as, *e.g.,* decrypting a ciphertext requires knowing the key) and a conclusion with a $\mathsf{K}^{\downarrow}$ fact. Construction rules have only $\mathsf{K}^{\uparrow}$ premises and a $\mathsf{K}^{\uparrow}$ conclusion. Accordingly, the Out rule produces a $\mathsf{K}^{\downarrow}$ conclusion, while In requires $\mathsf{K}^{\uparrow}$ premises. The transition from $\mathsf{K}^{\downarrow}$ to $\mathsf{K}^{\uparrow}$ is achieved by a special "Coerce" rule, but no direct conversion from $\mathsf{K}^{\uparrow}$ to $\mathsf{K}^{\downarrow}$ is possible, preventing loops. This enforces deconstruction before construction.[15] The resulting normal deduction rules are depicted in Figure 2.9, including the usual pairing and projection operators. With these rules, the adversary avoids redundancy as shown in Figure 2.8.

15: Note that one can still use constructor before deconstructor rules to construct "auxiliary" messages. For instance the decryption rule:

$$\frac{\mathsf{K}^{\downarrow}(\mathsf{senc}(x, y)) \quad \mathsf{K}^{\uparrow}(y)}{\mathsf{K}^{\downarrow}(x)}$$

forbids constructing the ciphertext $\mathsf{senc}(x, y)$ before decrypting it ($\mathsf{K}^{\downarrow}(\cdot)$), but allows all combinations of rules on the key y ($\mathsf{K}^{\uparrow}(\cdot)$).

Normal Form Conditions *Normal dependency graphs* are dependency graphs that satisfy all the 12 normal form conditions detailed in [DKK26], which are designed to exclude redundant deductions, while keeping all the relevant traces.[16] We do not formally detail these conditions here —referring the curious reader to [Sch+12a; Sch12; Dre+17]— and only list some of them, slightly simplified:

- **N1.** All terms from the dependency graph should be E'-normalized modulo AC ($\downarrow_E$).
- **N3.** If there are two conclusions c and c' with conclusion facts $K^{x}(m)$ and $K^{x'}(m')$ such that $m =_{AC} m'$ and $x = x'$ ($\uparrow$ or $\downarrow$), then $c = c'$.
- **N4.** All conclusion facts $K^{\uparrow}(f(t_1, \ldots, t_n))$ where f is an invertible[17] function symbol are conclusions of the construction rule for f.
- **N5.** If a node i has a conclusion $K^{\downarrow}(m)$ and a node j has a conclusion $K^{\uparrow}(m')$ with $m =_{AC} m'$, then $i < j$ and either the operator at the root of m is invertible or the node j is an instance of Coerce.

16: These normal form conditions were later extended to handle more expressive equational theories such as XOR and associative pairing, see Section 4.1, we refer to [DKK26] for the latest version of these conditions.

17: An operator $f \in \Sigma$ of arity n is invertible [Dre+17] if for all $i \in \{1, \ldots, n\}$, there exists a deconstruction rule $\mathsf{K}^{\downarrow}(f(t_1, \ldots, t_n)) \dashv[]\rightarrow \mathsf{K}^{\downarrow}(t_i)$.

Most of the conditions avoid redundancy and enforce that all terms are in normal form (**N1**). **N3** avoids multiple deduction of the same term. **N4** forbids deduction by coerce rule on invertible function symbols that instead must be created by construction. **N5** enforces the ordering of $\mathsf{K}^{\downarrow}$ before $\mathsf{K}^{\uparrow}$ facts.

It has been proven that the set of traces remains the same when restricting to normal dependency graphs, which allows Tamarin to restrict the search space to these graphs without losing any relevant trace.

Theorem 2.4.3 (See [Sch12, Lemma 3.19] or [DKK26, Theorem 1] for a formal statement and proof) *The set of traces obtained by all executions of all dependency graphs is the same as all the traces obtained by all executions of all dependency graphs in normal forms.*

2.4.1.3. Constraint Solving Procedure [Sch12; Mei13]

To construct normal dependency graphs and prove or disprove security properties —declared as lemmas—, Tamarin relies on *constraint solving*. A *constraint* represents an atomic condition on a dependency graph: the existence of a specific rule instance (I), of a certain edge (D), or a formula from a class of first-order formulas similar to the ones used in the definition of security properties (see Section 2.3.3). A *constraint system* Γ is a finite set of constraints. A P-solution of Γ is a normal dependency graph dg for a protocol P such that there exists a substitution θ with $dg\theta$ satisfying all constraints of Γ (see [Sch+12a] for the formal definitions).

The *constraint reduction relation* $\rightsquigarrow_P$ is defined between constraint systems and sets of constraint systems, where sets of constraint systems represent case distinctions. Intuitively, $\rightsquigarrow_P$ refines constraint systems: Tamarin's algorithm iteratively refines the initial constraint system (encoding the negation of the property to prove) until either a solved system is found (yielding a counterexample) or all refined systems contain contradictory constraints (proving the property). The constraint reduction rules are detailed in [Sch+12a]; their correctness is established by showing that they preserve the set of solutions [Sch12, Theorem 3.33] (or [Mei13, Theorem 4]), *i.e.,* $\rightsquigarrow_P$ is sound and complete.

Proof Guidance Tamarin provides two important proof-guidance mechanisms. *Helping lemmas* are previously verified lemmas declared as additional trusted constraints when proving subsequent lemmas, allowing the analyst to decompose a hard verification goal into sequential, tractable steps. Additionally, *tactics* and *oracles* let users supply heuristics that guide the constraint reduction order, prioritizing which open constraint to resolve first. All of these mechanisms were proven sound and complete in the sense that they do not rule out any relevant trace, they may, however, impact the efficiency and the (non-)termination of the algorithm.

2.4.2. The ProVerif Prover

ProVerif [Bla16] is very similar to Tamarin in terms of capabilities but relies on a radically different verification algorithm based on a *resolution-based saturation procedure* of *Horn clauses*. That said, the message model is essentially the same as Tamarin's and also relies on a term algebra with an equational theory.

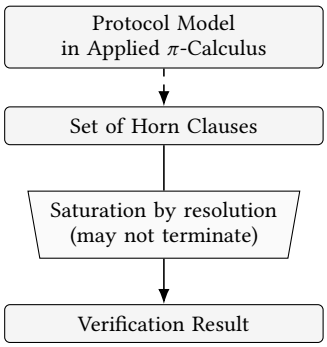


Figure 2.10.: ProVerif's verification workflow. →: translation or process is sound & complete. ⇢: translation is sound only (over-approximation). The saturation procedure also takes the security property to verify as input.

Protocol Model However, the protocol model is different and relies on a *process algebra* with replication and exchange of formal terms, called the *applied π-calculus* [AF01]; see an example in the margin. While the "input language" is different, it offers similar expressiveness as Tamarin's MSRs, and the two tools can be used to analyze roughly the same protocols and security properties.

Example (Continuing Example 2.3.5). Simplified ProVerif model of Sigma':

```
let processA(skA:skey, pkB:pkey) =
 new nx; new infoA;
 out(c, exp(g,nx));
 in(c, (yexp, yinfo));
 let k = exp(gy, nx) in
 event Start_A(pk(skA), pkB, k);   [...]

let processB(skB:skey, pkA:pkey) =
 new ny; new infoB;
 in(c, (xexp, xinfo));
 let k = exp(gx,ny) in
 event Start_B(pkA, pk(skB), k);
 out(c,(exp(g,ny), infoB));   [...]
```

`new` generates fresh names; `out`/`in` send/receive on the public channel `c`; `event` is analogous to Tamarin action facts.

Formally, the protocol —encoded as processes in the applied π-calculus— is translated into a set of *Horn clauses*. Adversary deduction rules are also encoded as Horn clauses, for instance with clauses like $\text{attacker}(x) \wedge \text{attacker}(y) \rightarrow \text{attacker}(\text{exp}(x, y))$.

The translation of the protocol is proven sound but is not complete since well-documented over-approximations are used. One fundamental source is the treatment of *nonces* (fresh names generated by `new`): Horn clause logic has no native notion of freshness, so ProVerif cannot express

that a nonce is distinct from all previously generated values. Instead, each nonce is abstracted as a *function of the current context* —notably the messages received by the process up to that point— making two nonces generated at the same program point indistinguishable whenever their contexts coincide. This abstraction is sound (any real attack is captured by a Horn clause derivation) but not complete: the prover may derive facts that rely on two contextually identical nonces being equal, even though they would be fresh and distinct in any real execution. Another notable source of over-approximation is ProVerif's treatment of *tables* (mutable key-value stores, declared with table and accessed via insert/get), which ProVerif uses to model persistent stateful storage. Because the Horn clause encoding cannot track the exact dynamic contents of a table, ProVerif merges all possible table states, potentially allowing lookups of entries that could not co-exist in any real execution.

As a consequence, ProVerif may produce two kinds of inconclusive results: (i) *non-termination* or (ii) *cannot be proved*, where a property is reported as violated in the Horn clauses model but ProVerif failed to determine if it is a real attack on the protocol model as well. In both cases, ProVerif may require manual proof guidance to obtain a conclusive result, which we mention below.

Example (Horn clauses for Sigma'). The processes above yield several clauses. *From A's first output:* $C_1 := \rightarrow$ attacker(exp($g, n_x[\,]$), infoA[ ]). This clause has no hypothesis and $n_x[\,]$ is a nonce abstraction as an operator over the current context, here empty; similarly for infoA[ ]. *From B's first input and output:* $C_2 :=$ attacker(xexp) $\wedge$ attacker(xinfo) $\rightarrow$ attacker(exp(g,n_y[xexp,xinfo]), infoB[xexp,xinfo]). The nonce abstractions n_y and infoB depend on received xexp and xinfo, so they differ for different inputs.

Security Property Model As in Tamarin (see Section 2.3.3), the security properties are encoded as first-order formulas (called *queries*) over so-called *events*, which are similar to Tamarin's *action facts*.

ProVerif also supports two property types analogous to those of Tamarin. *Lemmas* are properties that ProVerif proves first and then reintroduces as hypotheses when verifying subsequent queries; they play the same role as Tamarin's helping lemmas and allow breaking a hard verification problem into smaller, tractable steps. *Axioms* are formulas asserted unconditionally, without proof. Conceptually, they play the same role as Tamarin's *restrictions*: they constitute a useful modeling technique by ruling out traces that (should) correspond to impossible or irrelevant executions in the real world.

Example. The authentication property of B from A's point of view is roughly written as in Tamarin (see Example 2.3.8) assuming an event $\mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ is triggered at the end of A's process. ProVerif translates this into the task of deriving the *violation goal*: find $\mathsf{pk}_A, \mathsf{pk}_B, k$ such that $\mathsf{end}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ is derivable from the Horn clauses while $\mathsf{start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ is not. If every resolution path for end_A always requires a precedent $\mathsf{start}_\mathsf{B}$, the saturation closes and the property is proved, otherwise an attack derivation is found. ProVerif then tries to reconstruct a concrete attack trace from the attack derivation from Horn clauses, which may fail if the attack relies on the over-approximations mentioned above.

ProVerif Verification Algorithm ProVerif's verification algorithm does not rely on constraint solving to realize a backward search. Instead, it relies on saturation-based reasoning: the protocol's Horn clauses are saturated by a dedicated *resolution algorithm*, which derives all consequences of the protocol specification. Whether a security property holds is then determined by checking if a formula representing a property violation and the saturated set are inconsistent, *i.e.,* admit no solution.

Finally, ProVerif also supports the verification of some form of equivalence verification by the means of diff-terms as in Tamarin (see Section 2.3.3). Roughly speaking, ProVerif translates the equivalence of two processes P and Q into a reachability property of a *biprocess* that explores executions of P and Q in parallel in lock-step. This over-approximating translation is sound but not complete, and may fail to prove an equivalence even when it holds. Tamarin's approach to equivalence verification is similar and suffers from the same limitation. We revisit this problem in Section 4.2.2.

Proof Guidance Crucially, the resolution procedure is guided by a *selection function* that determines which facts within a clause are resolved in priority. By default, ProVerif uses built-in heuristics for this selec-

tion, but users can influence it to avoid non-termination and false attacks —a mechanism we will leverage in Section 5.6. Concretely, three main guidance mechanisms are available. First, noselect (respectively nounif) annotations on individual facts (atoms of Horn clauses) instruct the selection function never to select a specified fact for resolution (respectively for being unified with another selected fact). These can prevent the resolution from following unproductive derivation paths that would otherwise loop and prioritize other facts that can lead to a proof or a counterexample. Second, the user can provide explicit name contexts to modify the translation to Horn clauses for these names, which allows fine-tuning the trade-off between precision (less false attacks with larger contexts) and efficiency of the resolution procedure (more efficient and less prone to non-termination with smaller contexts). Third, users can supply *helping lemmas*: auxiliary security properties that ProVerif proves first (under the same Horn clause model) and then reintroduces as additional hypotheses; these can prune derivations that would otherwise reach a spurious attack conclusion or lead to non-termination. When used wisely, these techniques allow ProVerif to be very efficient and to handle a wide range of protocols and security properties. As for Tamarin, these proof guidance are in the user-space and are proven to preserve soundness, so they do not affect the trustworthiness of the results.

2.4.3. Other Approaches and Tools

We briefly mention other tools for protocol verification in the DY model; see [Bar+21a] for a comprehensive survey.

Unbounded Number of Sessions Maude-NPA [EMM09] handles rich equational theories (notably XOR and associative-commutative operators) more generically than Tamarin or ProVerif, but lacks support for global mutable state and else branches, and suffers from frequent termination problems [San+14]. DY* [Bha+21] is an interactive proof framework that scales to whole protocols but does not offer automated proof search. Sapic+ [Che+22] is a sound compiler from a language close to the applied π-calculus to both Tamarin and ProVerif input languages, enabling the use of both back-ends on the same model.

For a Bounded Number of Sessions Bounding the number of sessions enables decision procedures even for equivalence properties; symbolic execution with constraint systems or dedicated resolution-based saturation procedures handle the remaining infinite branching due to attacker-chosen inputs. The Spec prover [TD10] decides *open bisimulation* for linear roles[18], restricted to symmetric encryption and pairs. AKISS [Cha+16] handles richer equational theories (convergent rewriting systems with the FVP) via Horn-clause saturation, was first restricted to linear roles and then extended to handle else branches [GK17]. APTE [Che14], a precursor of DeepSec, was the first tool to decide trace equivalence for protocols with else branches. DeepSec [CKR18] supersedes it, deciding trace equivalence for a rich fragment of the applied π-calculus including else branches, private channels, and richer equational theories. We revisit this line of work in Section 4.2.1.

18: Linear role means without "else branches".

Part A.

Advancing Formal Methods for Cryptographic Protocol Security Verification

Contents of this Part

3 New Modeling Techniques 41

3.1 Hash Functions Weaknesses in the Real World 42

3.2 Threat Models for Weak Hash Functions 43

3.3 Automated Analysis with Weak Hash Functions 46

3.4 Analyses Results 48

3.5 Other Modeling Technique Advances 49

4 Improving Verification Tools 52

4.1 Equational Theories 52

4.2 Equivalence Verification 55

5 New Proof Methodologies 58

5.1 Dealing with a Large Collection of Verification Goals 59

5.2 Defining a Lattice of Verification Goals 61

5.3 Static Analysis 64

5.4 Lattice Exploration 65

5.5 Example of Results 67

5.6 Proof Helper Methodology for ProVerif 68

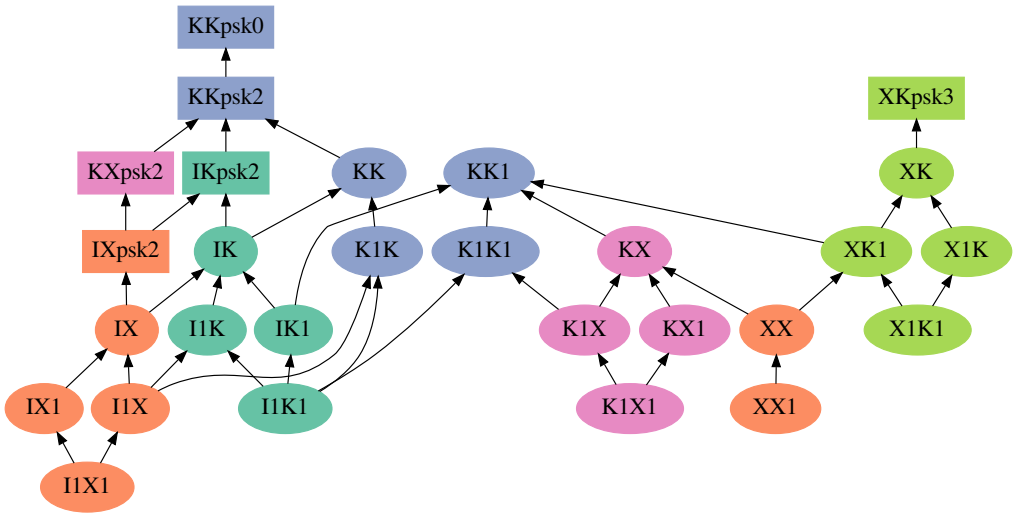


Figure 2.11.: This figure depicts one of the results of our formal analysis of the Noise family of protocols presented in Chapter 5. Each node of the depicted lattice represents a protocol from this family and the arrows specify a partial order relating a protocol to another one with strictly stronger secrecy, agreement, and anonymity guarantees. To obtain such results, we developed new analysis methodologies that aim at *measuring* to what extent a protocol is secure, rather than just proving or disproving a single security goal for a single protocol instance. (Here, we restrict the list of protocols to those for which the initiator and recipient have a static key, colors indicate identical system parameters; see Section 5.5.)

The initial steps in the formal analysis of a security protocol involve modeling the cryptographic primitives and the protocol itself. This process inherently involves abstractions and simplifications, which can create a gap between models and real-world implementations. Such a gap sometimes hides attacks that exploit properties not captured in the model.

Advancing Modeling Techniques. In our previous works, we have investigated some of these gaps and proposed solutions to narrow them down. Chapter 3 emphasizes our new modeling techniques, and mostly focuses on one of them where we modeled weak hash functions in security protocols. We also briefly mention our other modeling techniques for distributed cryptographic protocols and for protocols using identity-based cryptography.

Advancing Verification Algorithms and Tools. Once a protocol and its security goals are modeled, the next step is to verify whether the protocol satisfies these goals using automated verification tools. In the last two decades, significant progress has been made in the development of such tools and their underlying verification algorithms. We have contributed to this progress by addressing some limitations of existing verification algorithms for equiavelence-based properties —required to model most privacy properties— which resulted in net improvements in terms of efficiency (scalability) and expressiveness (scope of analysis). We briefly present these contributions in Chapter 4 both for the setting of an unbounded number of sessions and for the setting of a bounded number of sessions.

Advancing Verification Methodologies. Finally, even when verification is in principle possible, practical challenges may arise when analyzing complex, real-world protocols. First, symbolic verifiers allow user-provided proof hints and heuristics to guide the proof search, which are sometimes necessary for the verification to succeed. We present a proof methodology to tackle such challenges in ProVerif that we developed in the context of a particular complex protocol analysis [*DHK25a*] —largest ProVerif model to date— and that is of independent interest. Second, being able to

analyze a few a-priori known security goals for a single protocol instance is often insufficient in practice. In line with our objective to use symbolic analyses to explore and *measure* the *level of security* provided by a protocol or a family of protocols, we have developed new proof methodologies to tackle these challenges. We present these contributions in Chapter 5.

Beyond presenting these individual contributions, this chapter also emphasizes the process of analyzing cryptographic protocols using formal methods: (i) modeling, (ii) leveraging verification tools, (iii) performing the analysis. Interestingly, some of our works contributed to all three of these steps. For instance, our work on analyzing cryptographic protocols with weak hash functions [*Che+23*] required:

(i) New modeling techniques for hash functions (Chapter 3);
(ii) Extension of the verification algorithms and tools (ProVerif and Tamarin) to expand the supported equational theories to capture these new models (Section 4.1.2);
(iii) Improving analysis methodology by developing new goals exploration techniques (Section 5.1.2).

Similar multi-level contributions were also required for our analysis of the 4G and 5G mobile communication protocols presented in Chapter 7 and the OPC UA protocol presented in Chapter 6, both in Part B.

3. New Modeling Techniques

🔍 *Focus on Modeling Weak Hash Functions [Che+23]*

Cryptographic hash functions are a fundamental building block in nearly all cryptographic protocols. The main three desirable properties that a cryptographic hash function (denoted hereafter by $\mathsf{H}(\cdot)$) should satisfy are well-established: first and second preimage resistance, and collision resistance [MOV96], stating that there is no better algorithm than brute force to

- **(Preimage resistance)** find x given h, such that $h = \mathsf{H}(x)$;
- **(Second preimage resistance)** find y given x, such that $\mathsf{H}(y) = \mathsf{H}(x)$;
- **(Collision resistance[1])** find x, y such that $\mathsf{H}(x) = \mathsf{H}(y)$.

Collision-resistance implies second preimage resistance, as finding a second preimage effectively results in a collision.

However, when developing proofs of security protocols that use hash functions, it turns out using the three resistance properties as assumptions is very complex and error-prone. Because of this, the **Random Oracle Model (ROM)** was introduced in 1993 by Bellare and Rogaway [BR93] as a simpler proof methodology. Intuitively speaking, the Random Oracle Model (ROM) models "perfect" hash functions: as functions whose outputs are chosen uniformly at random, independent of the input. The ROM satisfies first and second preimage resistance and collision resistance. It thus has the most desirable cryptographic hash properties, effectively over-approximating the security of real-world hash functions. Therefore, proving that a protocol is secure using the ROM does not guarantee that it is secure when instantiated with a real hash function. However, the vast majority of protocol proofs use the ROM due to its simplicity.

In practice though, real hash functions are unfortunately far from perfect. Widely deployed hash functions suffer from known imperfections. For instance, SHA1 is considered to be weak or broken with respect to collision or preimage resistance but no efficient known attack affects second-preimage resistance. SHA2 is currently still considered secure, but it is not ideal in a sense that it allows the so-called "length extension attacks", which can enable someone to compute[2] $hash(x\|y)$ even if they do not know x – which is not possible with a perfect hash function or in the ROM.[3] As we will see, this enables *length-extension attacks* in certain protocol contexts—a possibility known since at least 1992 [Tsu92].

Several factors contribute to the gap between real-world hash functions and their idealized version. First, hash function security cannot be reduced to hard problems and is instead based on heuristics, and history shows that many once-trusted hash functions were later broken [Aur17; Wil17]. Second, it was previously assumed that hash function weaknesses would not affect protocols as long as (second) preimage resistance holds [BL16; Ste21], but this has proven false. Third, even hash functions meeting standard requirements may permit attacks such as length extension. We thus have to face the reality: protocols use hash function that are already imperfect[4], and history has shown that over time, hash functions that appear secure now will become easier to attack in the future.

This raises the natural question:

3.1 Hash Functions Weaknesses in the Real World 42
3.2 Threat Models for Weak Hash Functions 43
3.2.1 Example: Hash Transcript Collisions and Chosen-Prefix Collision (CPC) . . 43
3.2.2 Lattice of Adversarial Capabilities 44
3.3 Automated Analysis with Weak Hash Functions . 46
3.4 Analyses Results 48
3.5 Other Modeling Technique Advances 49
3.5.1 ID-Based Cryptographic Protocols 49
3.5.2 Modeling Distributed Cryptographic Protocols . 50

1: By the pigeon hole principle, collisions necessarily exist. Therefore, collision-resistance is only informally defined as the absence of a *known* algorithm to find a collision, faster than the generic birthday search-based brute-force algorithm.

2: The operator $\|$ denotes message concatenation subject to an associative equational theory: $(x\|y)\|z = x\|(y\|z)$.

3: On length-extension: Many deployed hash functions are based on the Merkle-Damgård (MD) construction: $\mathsf{H}(m\|m') = f(\mathsf{H}(m), m')$ where f is the underlying compression function. This design enables incremental hashing: $H(m\|m')$ can be computed from a compact intermediate state based on m, without storing both m and m' in memory. As a consequence, MD constructions satisfy the *length-extension* property: given $H(m)$ and m', one can compute $H(m\|m')$

4: See Table 3.1 and Section 3.1 for concrete examples.

How can we check if a protocol using a hash function with a particular weakness meets its security guarantees?

History has shown such attacks can be rare but very subtle, *e.g.,* [SLW07; Ste21; BL16], and thus difficult to detect manually.[5] As we advocated throughout this manuscript, automated protocol analysis tools seem a promising approach to systematically find such attacks. However, they model ideal hash functions only, intuitively in the ROM, *i.e.,* traditionally as an operator in a free term algebra modeling. Thus, like computational proofs that use the ROM, such analyses miss any attacks that exploit the use of an imperfect hash function.

Contributions. In [*Che+23*], we revisit cryptographic hash function definitions, common weaknesses, and the potential attacker capabilities that arise from them. Based on this, we develop a methodology to systematically discover attacks on protocols that exploit their use of "less-than-perfect" hash functions, and show how this can be implemented in ProVerif and Tamarin through advanced features and tool extensions. Our methodology can be used in the design phase to avoid the use of hash functions that are too weak, or to find and fix problems in deployed protocols.

5: An interesting example is a forgery attack based on the length-extension property that affected Flickr in 2009 [Duo09]. A more subtle class of attacks is so-called *hash transcript collisions* attacks that affected TLS 1.2, SSH, and IKEv2 [BL16] (see Section 3.2).

Example 3.0.1 To formally model a hash function under the ROM, we consider the operator hash(·) without equation. Therefore, if t_1 and t_2 are different then hash(t_1) and hash(t_2) are different as well, that is the hash function has no collision. ◊

[*Che+23*]: Cheval, Cremers, Dax, et al., *Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses*, USENIX'23

3.1. Hash Functions Weaknesses in the Real World

In Table 3.1 we review a selection of widely used hash functions, the complexity of the best known collision and preimage attacks against them, and whether length-extension is possible. The conclusion is clear: high-profile cryptographic protocols still use hash functions that suffer from weaknesses that contradict the usual idealized security assumption (ROM) and even (second) preimage and collision resistance. We also see that hash functions get weaker over time as the attacks get more and more efficient (see also the survey [Ste21] and [Aur17]). Moreover, it is likely that hash functions that are deemed secure today will be weakened in the future.

The length extension property of many hash functions can theoretically be used to break a protocol's security, because it enables the following behaviors: *(i)* collisions can be extended since $\mathsf{H}(x) = \mathsf{H}(y)$ implies $\mathsf{H}(x\|s) = \mathsf{H}(y\|s)$ for any s, and *(ii)* the adversary can extend the payload under known hash outputs: given $\mathsf{H}(x)$, it can compute $\mathsf{H}(x\|s)$ for any known s. As an example of the latter, if the prefix x contains a shared secret, and the protocol relies on this to authenticate hash values, then the adversary can "forge" hashes by extending any hash values it observes. An early example of a widely deployed protocol that was vulnerable due to such an attack was Flickr in 2009 [Duo09]; we revisit and automatically find this attack in [*Che+23*].

As aforementioned, it was thought for a long time that cryptographic protocols are likely to remain secure even though they rely on weakened hash functions as long as the hash functions are (second) preimage resistant [BL16; Ste21]. For instance, even if the adversary can compute *some* c, c' such that $\mathsf{H}(c) = \mathsf{H}(c')$, which breaks collision-resistance, it seems unlikely that it can impact honest agents in a protocol who will compute hashes for inputs that are unrelated to c, c'. Unfortunately, this is a false sense of security: it has been shown that cryptographic protocols can be entirely broken when using hash functions that merely suffer from

Table 3.1.: Examples of widely used hash functions that are currently deployed in security protocols and do not offer perfect (random-oracle like) guarantees. The numbers indicate the complexity of the currently best known attack on the property [LP20; SA09; Sas+07; XLF13; ZL12]. For the hash functions currently deemed secure the best known attacks would be a brute-force approach; *e.g.,* the complexity to break collision resistance on SHA2-256 is 2^{128}. Crucially, this situation is not constant, but expected to get worse: history suggests that the numbers for the best attacks are likely to decrease over time for all hashes, see *e.g.,* [Aur17; Ste21]. Legend: ✓ = currently still secure, ⊗ = weak but no full attack yet, ✗ = known attack. * Theoretical attacks on (second) preimage resistance were found [ZL12; SA09], but they are still not feasible. ** No known attack but the small bit size makes a collision attack plausible. *** Ethereum uses the original Keccak-256, which differs from the NIST-standardized SHA3-256 in its padding.

Hash function	Year	Examples of currently deployed applications	Collision resistance	(2nd) Preimage resistance	No Length-extension
MD4	1990	NTLM key derivation for Microsoft Windows	✗ 2^{1}	⊗* 2^{95}	✗
MD5	1992	File checksums (md5sum)	✗ 2^{18}	⊗* 2^{123}	✗
SHA1	1995	Europay Mastercard Visa (EMV), File checksums, Telegram (< 2017)	✗ 2^{61}	✓ 2^{160}	✗
RIPEMD-160	1996	Bitcoin	⊗** 2^{80}	✓ 2^{160}	✗
SHA2-256	2001	Bitcoin, TLS, SSL, SSH, S/MIME, IPSec, DNSSEC, Linux/Unix password hashing, Telegram (> 2017)	✓ 2^{128}	✓ 2^{256}	✗
SHA2-512	2001	TLS, SSL, SSH, S/MIME, IPSec, DNSSEC, Linux/Unix password hashing	✓ 2^{256}	✓ 2^{512}	✗
SHA3-256	2012	Ethereum***	✓ 2^{128}	✓ 2^{256}	✓

some restricted classes of collisions [Ste21; SLW07]; see an example in Section 3.2.

Unfortunately, it is difficult and error-prone to manually assess if a cryptographic protocol can be broken if its hash function is vulnerable to some restricted class of collisions.

3.2. Threat Models for Weak Hash Functions

We have developed the first systematic, automated methodology to find protocol attacks that exploit weaknesses of real-world hash functions. At a technical level, we achieve this by symbolically modeling cryptographic weaknesses (*i.e.,* the lack of desirable cryptographic properties) as well as real-world attack classes that are not captured by classical security definitions for cryptographic hash functions.

3.2.1. Example: Hash Transcript Collisions and CPC

Using *hash transcript collisions*, we exemplify how certain collisions can be weaponized against protocols. They have been shown to affect various authentication protocols such as TLS 1.2, SSH, or IKEv2 [BL16]. For illustration, we continue with the example of the Sigma' protocol, introduced in Section 2.2.1 and depicted in Figure 2.1. Note that in practice info_A and info_B have variable length and are thus length-prefixed. As noted in [BL16], the parties do not directly authenticate and agree on the transcript but rather on the hash of the transcript. If the hash function were perfect, this would not matter —but it makes a difference for real-world hash functions.

Machine-in-the-Middle (MIM) scenario First, one should note that the protocol is not immediately broken, even if the hash function H is not preimage resistant. Assume a MIM attacker: the attacker can replace messages m_1 and m_2 by messages $m_1' := g^{x'} \| \mathsf{info}_A'$ and $m_2' := g^{y'} \| \mathsf{info}_B'$ of

its choice. This results in the message

$$m_3 = \mathsf{sign}(\mathsf{sk}_A, \mathsf{H}(m_1 \| m_2')), \mathsf{mac}(g^{xy'}, \mathrm{A}).$$

If the attacker does not know sk_A, it cannot modify m_3. Hence, B will check whether $\mathsf{H}(m_1 \| m_2') = \mathsf{H}(m_1' \| m_2)$. Clearly, (second) preimage attacks do not directly allow such a MIM attack to succeed, as the input for the target hash output $\mathsf{H}(m_1 \| m_2')$ must be of the form $m_1' \| m_2$, where m_2 is fixed and not adversary-chosen. Similarly, the mere existence of collisions, say $c \neq c'$ such that $\mathsf{H}(c) = \mathsf{H}(c')$, cannot be used to break this protocol.

Hash transcript collisions attacks CPC [SLW07; Ste21] are among the least costly collisions to compute and yet can be weaponized against protocols. Given two prefixes, p_1 and p_2, a CPC attack computes two suffixes $s_1 \neq s_2$ such that $\mathsf{H}(p_1 \| s_1) = \mathsf{H}(p_2 \| s_2)$. When additionally $p_1 = p_2$, such a collision is called Identical-Prefix Collision (IPC) and is even less costly to compute.

As we shall see, Sigma' is entirely broken as soon as

1. the used hash function suffers from CPC attacks,
2. obeys the length-extension property, and
3. the length of m_2 is predictable.

Indeed, given m_1 sent by A, the adversary can choose arbitrary x', y' and compute a CPC for prefixes $m_1 \| g^{y'}$ and $g^{x'}$, resulting in suffixes $\mathsf{infoPartial}_B'$ and info_A' such that:

$$\mathsf{H}(m_1 \| g^{y'} \| \mathsf{infoPartial}_B') = \mathsf{H}(g^{x'} \| \mathsf{info}_A'). \qquad (\dagger)$$

Moreover, the claimed length field of $\mathsf{infoPartial}_B'$ can be chosen to be $|\mathsf{infoPartial}_B'| + |m_2|$ The MIM adversary then uses $m_1' := g^{x'} \| \mathsf{info}_A'$ and $m_2' = g^{y'} \| \mathsf{info}_B'$ where $\mathsf{info}_B' = \mathsf{infoPartial}_B' \| m_2$. By the length-extension property of H, we obtain by appending m_2 to the above collision (†):

$$\mathsf{H}(m_1 \| m_2') = \mathsf{h}(m_1 \| g^{y'} \| \mathsf{info}_B') = \mathsf{H}(m_1' \| m_2).$$

Therefore, the MIM adversary successfully impersonated A and B and hijacked the session key, *i.e.,* $g^{xy'}$ with A and $g^{x'y}$ with B. We depict the attack flow in Figure 3.1.

To give a sense of the attack cost, finding such a collision costs about 2^{39} for MD5 and $2^{63.4}$ for SHA1 [Ste21; LP20]. As we shall see in Section 3.4, other kinds of CPC but no IPC affect Sigma', findings we formally establish with our automated formal analysis framework.

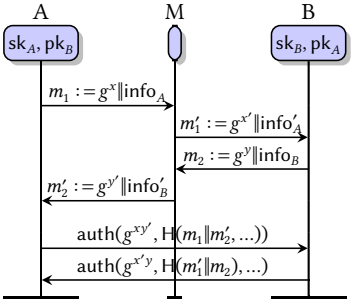


Figure 3.1.: CPC attack on Sigma': an adversary M forges m_1', m_2' so that $\mathsf{H}(m_1 \| m_2') = \mathsf{H}(m_1' \| m_2)$, hijacking the session key $g^{xy'} = g^{x'y}$. Here, $\mathsf{auth}(k, h, \ldots)$ denotes the signature and MAC exchange of Figure 2.1 using session key k and transcript hash h.

3.2.2. Lattice of Adversarial Capabilities

We now develop our hierarchy of hash function models. We start from the ROM model, which represents an ideal hash function, *i.e.,* for which the adversary has the least possible capabilities to manipulate or learn information from it. We then strengthen the adversary's capabilities in four main independent dimensions we have identified, corresponding to possible weaknesses of hash functions: collision-related weaknesses, length-extension style weaknesses, output-control weaknesses that can model, *e.g.,* backdoored hash functions, and weaknesses that leak information about the inputs from the output.

The overall structured lattice is depicted with its dimensions in Figure 3.2. Capabilities higher up represent stronger adversarial capabilities. For example, the bottom row, *i.e.,* threat model $\emptyset$, corresponds to the ROM model of an ideal hash function. Conversely, the strongest threat model in

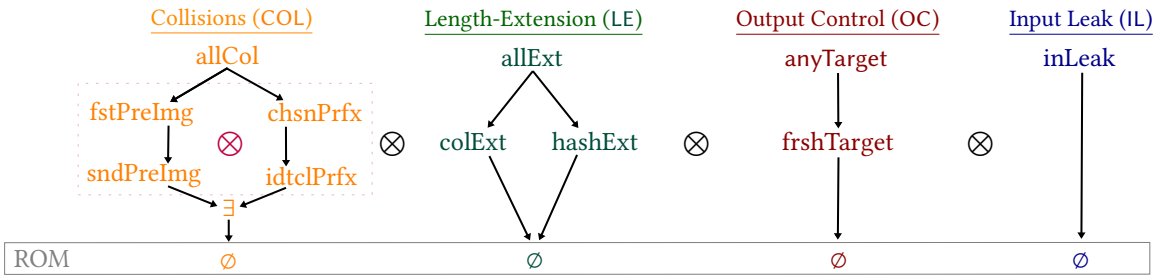


Figure 3.2.: Lattice of adversarial capabilities. An edge $x \to y$ expresses that x is a stronger capability than y. Each column is a dimension described at the top. Capabilities from different dimensions can be combined into threat models, *e.g.*,{idtclPrfx,allExt}. The minimal threat model is the empty one, ∅, which corresponds to the ROM.

which the adversary has all capabilities (and corresponds to modeling the weakest hash function) is {allCol,allExt,anyTarget,inLeak}. We now provide some more details for each dimension in turn.

Allowed Collisions (COL). Without ROM-like constraints, all kinds of worst-case collisions might be considered, provided that the resulting hash function is indeed a function. This includes for example the constant function that maps all inputs to a single value, or even a function that maps any input to some adversary-chosen output. A protocol can be deemed insecure with such strong adversarial capability due to unrealistic attacks. To refine this, we restrict the allowed choices of hash outputs to the relevant classes of collisions one may want to consider This is done using a **collision-relation** $\sim_c$: it captures that for all x and y such that $x \sim_c y$, the hash function H allows collisions $H[x] = H[y]$. We cover a large spectrum of types of collisions that can be combined as defined in Table 3.2 (the last two rows are explained next). To define those relations, we introduce a number of abstract operators $\mathsf{cp}_1(\cdot,\cdot)$, $\mathsf{cp}_2(\cdot,\cdot)$, $\mathsf{sp}_1(\cdot)$, $\mathsf{sp}_2(\cdot)$, $\mathsf{pi}^1(\cdot)$, $\mathsf{pi}^2(\cdot)$, $\mathsf{c}()$, $\mathsf{c}'() \in \Sigma$ that do not occur in protocols. Those operators correspond to computations performed by the adversary to find a certain collision or preimage, and are motivated by real-world attack strategies. For instance, given two messages p_1, p_2, the messages $\mathsf{cp}_1(p_1, p_2)$ and $\mathsf{cp}_2(p_1, p_2)$ correspond to the message the adversary obtains when computing a CPC for the prefixes p_1 and p_2: the hash of $p_1 \| \mathsf{cp}_1(p_1, p_2)$ equals the hash of $p_2 \| \mathsf{cp}_2(p_1, p_2)$, as expressed by $\sim_{\mathsf{CP}}$. (See for example the aforementioned CPC-based hash transcript collision attack from Section 3.2.1.)

Given a relation $\sim_c$ corresponding to the chosen kinds of collisions, the allowed hash outputs determined by the output control dimension OC (explained below) are filtered out.

Length-Extension (LE). The two last types of collisions defined in Table 3.2 are specific to the length-extension property and weaknesses of hash functions built with MD. The first, hashExt, captures the adversarial capability to extend the payload that is under a hash with some adversary-chosen suffix, and is captured by the collision-relation $\sim_{\mathsf{LEa}}$. Namely, given a hash output $H[x]$, for any suffix s of its choice the adversary can compute $H[x\|s]$ without knowing x. Indeed, the hash output of $x\|s$, which the attacker cannot compute, is allowed to collide with the one of $H[x]\|s$, which the attacker can compute. This is the most classical weakness of length-extension. The second, colExt, corresponds to the fact that collisions may be extended, *i.e.,* as soon as $H[x] = H[y]$, we will also have that $H[x\|s] = H[y\|s]$ for any s. Interestingly, the keyed-hash functions HMAC-SHA2 and HMAC-MD5 constructions have colExt but not hashExt for a given key.

Output Control (OC). In a worst-case scenario, we consider a hash function where the attacker may control the output of the hash function to some extent, provided that the resulting hash is indeed a function. The hash out-

Table 3.2.: Intuition behind the basic collision-relations $\sim_c$ depending on the chosen adversarial capabilities. Many of these relations define that there exist collisions that can be computed for very specific, but not all, input patterns. Collision-relations in different dimensions can be combined by taking their union.

Capability	$\sim_c$	Intuitions behind the types of allowed collisions
$\emptyset$	$\sim_\bot$	Ideal model in which hash outputs never collide; *i.e.,* $\forall t \neq t' : t \not\sim_\bot t'$
$\exists$	$\sim_\exists$	There exist two constants c and c' whose hashes collide $H[c] = H[c']$; *i.e.,* $c \sim_\exists c'$
fstPreImg	$\sim_1$	Given $o = H[t]$, the adversary can compute a preimage $t' = \mathrm{pi}^1(o)$ such that $H[t'] = o$; *i.e.,* $t \sim_1 \mathrm{pi}^1(H[t])$
sndPreImg	$\sim_2$	Given t, the adversary can compute a second preimage $t' = \mathrm{pi}^2(t)$ such that $H[t'] = H[t]$; *i.e.,* $t \sim_2 \mathrm{pi}^2(t)$
chsnPrfx	$\sim_{\mathsf{CP}}$	Given t, t', the adversary can compute $u = \mathrm{cp}_1(t,t')$ and $u' = \mathrm{cp}_2(t,t')$ such that $H[t\|u] = H[t'\|u']$; *i.e.,* $t\|u \sim_{\mathsf{CP}} t'\|u'$
idtclPrfx	$\sim_{\mathsf{IP}}$	Given t, the adversary can compute $u = \mathrm{sp}_1(t)$ and $u' = \mathrm{sp}_2(t)$ such that $H[t\|u] = H[t\|u']$; *i.e.,* $t\|u \sim_{\mathsf{IP}} t\|u'$
allCol	$\sim_\top$	All hash outputs can collide, which models the worst possible collision case; *i.e.,* $\forall t, t' : t \sim_\top t'$
hashExt	$\sim_{\mathsf{LEa}}$	Length-extension collision. Given $H[x]$ and s, the adversary can compute $H[x\|s]$; *i.e.,* $x\|s \sim_{\mathsf{LEa}} H[x]\|s$
colExt	$\mathsf{LEc}(\sim_c)$	Length-extension closure. $H[x\|s]$ collides with $H[y\|s]$, if $H[x] = H[y]$ (based on any of the previous capabilities).

puts can be taken from *(i)* a set of atoms $\mathscr{A}_H$ defined a subset of names $\mathscr{A}$ otherwise not used in protocol models, that model fresh values unknown to the attacker (unless revealed), as in the ROM, *(ii)* fresh atoms chosen by the attacker (frshTarget), or *(iii)* arbitrary messages provided that the adversary knows them (anyTarget). The second capability frshTarget models the case where the attacker can partially control the outputs of the hash function that nonetheless must be names. This captures some type-flaw attacks but the attacker cannot control the actual shape of the hash values which will appear as junk bytes. The third one anyTarget additionally captures arbitrary type-flaws attacks, where the attacker can fully control the hash output to an arbitrary value.

Input Leak (IL). Finally, hash functions are sometimes implicitly assumed to guarantee confidentiality of their inputs, and sometimes ill-used for this purpose in practice. In reality, some badly designed hash functions might leak information about their inputs. Previous symbolic analyses did not capture this capability, which can yield practical attacks. We model the worst case scenario in which the adversary can obtain the complete input with inLeak.

3.3. Automated Analysis with Weak Hash Functions

We automate our methodology in the two leading automated protocol analysis tools, Tamarin and ProVerif. To achieve this, we: (a) propose dedicated modeling techniques, and (b) extend both tools with new required features that are of independent interest beyond this work.

Capabilities modeled as operators Using an equational theory allows us to naturally and efficiently explore the set of scenarios corresponding to the capabilities fstPreImg, sndPreImg, $\exists$, as well as inLeak. Each threat scenario for a given dimension can be encoded by the following corresponding equation that we explain below.

$$\begin{array}{rlrl} fstPreImg & \mathsf{H}(\mathrm{pi}^1(z)) = z & \exists & \mathsf{H}(c) = \mathsf{H}(c') \\ sndPreImg & \mathsf{H}(\mathrm{pi}^2(z)) = \mathsf{H}(z) & inLeak & \mathsf{i}(\mathsf{H}(z)) = z \end{array}$$

We note that the absence of a weakness for each dimension is the default in symbolic models and does not require any particular modeling. For each threat model we simply add the corresponding equations for the collision and input leak dimension.

To express that a hash function is not *preimage resistant* (fstPreImg) we provide the attacker with an explicit operator $\mathsf{pi}^1(\cdot)$ that allows the attacker to compute a preimage of a hash z, *i.e.,* a value $\mathsf{pi}^1(z)$ that hashes to z. The absence of *second preimage resistance* (sndPreImg) is similarly expressed by the operator $\mathsf{pi}^2(\cdot)$. The absence of *collision resistance* (∃) relies on two distinguished constants c and c' on which the hash function collides. These do not have any particular structure and are not attacker chosen, modeling that a collision merely *exists*. Finally, we model that a hash function may leak (part of) its input (inLeak) by giving the attacker the capability to inverse the function using the operator $\mathsf{i}(\cdot)$, and no leak being the default does not require an equation.

We encoded this part of the hierarchy in both Tamarin and ProVerif, and used it on multiple case-studies. While such an equation-based model is easy to deploy using symbolic tools, it is also rather weak (and not a very effective at finding attacks): the equations are basically obtained by negating the security assumption and model the existence of a collision, or (second) preimage without giving the attacker any additional control. (We exemplified this gap with Sigma' in Section 3.2.1.) On the other hand, finding an attack with this model generally translates directly to a missing assumption on the security of the used hash function, instead of a practical attack.

Challenges with modeling associative ‖ and MD To go beyond the above *existential* modeling of weaknesses in hash function we ought to faithfully model the associative ‖ operator and of the MD construction whose properties can be exploited by an adversary (length-extension). This is required to model IPC, CPC, and length extension attacks, which are not covered using the aforementioned equations. Indeed, recall that given prefixes, p_1 and p_2, a CPC attack computes suffixes s_1, s_2 such that $\mathsf{H}(p_1 \| s_1) = \mathsf{H}(p_2 \| s_2)$. If, for example, a protocol participant computes the transcript $\mathsf{H}(m_1 \| m_2 \| m_3)$ and the attacker controlled parts are m_2 and m_3, then the suffix in the previous equation needs to be $m_2 \| m_3$. With a non-associative concatenation operator ($\langle \cdot, \cdot \rangle$ defined in Example 2.3.2) the CPC attack would fail as $\mathsf{H}(\langle\langle m_1, m_2\rangle, m_3\rangle)$ would not be identified with $\mathsf{H}(\langle m_1, \langle m_2, m_3\rangle\rangle)$. This raises a challenge for the existing tools, as neither Tamarin nor ProVerif were capable to model such an *associative operator*. We extended both tool algorithms and tools to support such operators, as detailed in Section 4.1.

paragraphModeling Advanced Threat Models with Restrictions Next, instead of giving the precise semantics for the hash function, we record all the calls to the operators modeling the hash function (by the means of *action facts*[6]) and discard execution traces, by the means of so-called *restrictions*[7] (see an example below), that violate the properties of the hash function that hold in the chosen threat model.

Example 3.3.1 We can represent a simplified threat model of {chsnPrfx,colExt} with a restriction over $\mathsf{Hash}(i, o)$ action facts recording that the hash function outputs o on input i. Such a restriction is a logical formula that requires that if a collision occurs for i and i', then it must be a length-extended CPC: the $\mathsf{cp}_i(p_1, p_2)$ represent the attacker chosen suffix for prefixes p_1 and p_2 (chsnPrfx) and l is a length extension on both

6: These log some rule applications in the execution traces in Tamarin; see Section 2.3.2 and example 2.3.5. In ProVerif, we use *events* for the same purpose.

7: Restrictions are Tamarin logical formulae considered as part of the specification and where any execution that does not satisfy it is discarded. In ProVerif, we use *axioms* for the same purpose.

Threat Models				Authentication
COL	LE	OC	IL	
fstPreImg,chsnPrfx	hashExt	anyTarget	inLeak	✓
allCol				✗
fstPreImg,idtclPrfx	allExt	anyTarget	inLeak	✓
chsnPrfx	colExt			✗

Table 3.3.: Sigma' security analysis. ✓: security holds, ✗: attack found.

inputs (colExt), which is formally written as[8]:

$$\forall i, i', o.\ \mathsf{Hash}(i, o)\ \&\ \mathsf{Hash}(i', o)\ \&\ i \neq i' \Rightarrow \exists p_1, p_2, l.\quad (\ i = p_1 \| \mathsf{cp}_1(p_1, p_2) \| l\ \&\ i' = p_2 \| \mathsf{cp}_2(p_1, p_2) \| l)$$

8: We omit time points and quantification for readability.

◊

Systematic exploration of the lattice We developed a Tamarin library that allows verification of a specified threat model in the lattice of hash weaknesses. To automate a systematic exploration of the full lattice of threat models, we developed a Python program that computes the set of all minimal, respectively, maximal scenarios that invalidate, respectively, validate the security goals. We will provide more details on this when covering lattice exploration strategies in Chapter 5. Using this verification methodology, that we automate, we obtain a comprehensive view of the security of a protocol with respect to hash function weaknesses. We show an example of a fully automatically generated table for Sigma' in Table 3.3 (detailed in Section 3.4).

3.4. Analyses Results

We apply our methodology to over 20 protocols, automatically rediscovering all previously reported attacks on those protocols that exploit weak hash functions, as well as finding several new variants. The analysis outcomes are twofold. First, our analysis confirms known attacks but also finds new variants of attacks under different threat models, demonstrating the practical relevance of our systematic methodology. We also found new attacks that were previously unknown, notably on IKEv2 and SSH[9]. Those attacks are valid in the considered threat models but often require a significant computational effort, *e.g.,* a CPC attack on SHA-1. However, they highlight protocol flaws or missing security assumptions in the analyzed protocols. We defer the details of the case studies to [*Che+23*] as we focus here on the methodology.

9: Regarding one of the attacks we found on the SSH protocol (*colliding input attacks*): We have later exchanged with colleagues who have found some data type definitions in RFC4251 (not the RFC4253 dedicated to the SSH protocol) that would make this attack impractical except for implementations with liberal parsing.

Second, for all affected protocols, our methodology allows us to precisely delineate the exact assumptions on the hash function, among the ones from our lattice, required for establishing a symbolic proof, which were previously unknown.

Example 3.4.1 As an example, consider the last row of Table 3.3: the protocol is broken if CPCs are possible (chsnPrfx) and can be extended thanks to colExt, even when hash outputs are completely fresh values. However, without either colExt or chsnPrfx, the protocol is deemed secure (otherwise this threat model would not represent a minimal violation). Inspecting the attack trace returned by Tamarin for this threat model, we observe that it corresponds to the CPC attack from [BL16] described

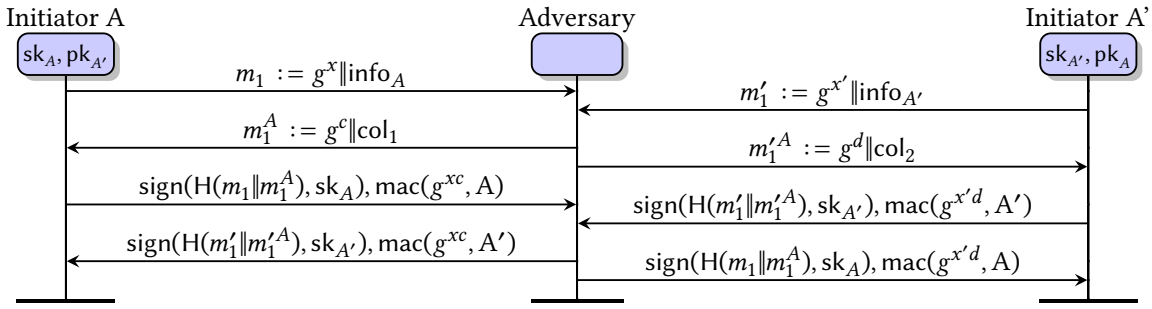


Figure 3.3.: The new CPC attack we found on Sigma'. $\mathrm{col}_i = \mathrm{cp}_i(L_1, L_2)$ for $i \in \{1,2\}$ where $L_1 := g^x \| \mathrm{info}_A \| g^c$ and $L_2 := g^{x'} \| \mathrm{info}_{A'} \| g^d$. Therefore $\mathrm{H}(m_1 \| m_1^A) = \mathrm{H}(m_1' \| m_1'^A)$ so that the signatures of A's and A' can be reused.

in Section 3.2.1. ◊

Example 3.4.2 More interestingly, we also automatically found a novel CPC attack on Sigma'. The attacker acts as a MIM in-between two agents A and A' both acting as initiator. It is able to impersonate A towards A', who believes A is acting as a responder, and A' towards A, who believes A' is acting as a responder. The adversary additionally learns both session keys. This attack thus violates session key secrecy, authentication, and agreement on the agents' role. The attack is depicted in Figure 3.3. A simple fix for this attack is to include the role name (initiator, responder) under the signature to avoid such role confusion. We show automatically that the attack disappears with this fix. We refer the reader to [*Che+23*] for more details on this attack and the other ones we found. ◊

3.5. Other Modeling Technique Advances

We briefly present here some other modeling advances that we have made, focusing on two specific contexts: identity-based cryptography and distributed cryptographic protocols, and omitting other advances such as our modeling of Quantum Key Distribution (QKD) protocols [*Hir19*].

3.5.1. ID-Based Cryptographic Protocols [*BHS19*]

In *identity-based cryptography* [Sha84], public keys are derived from participants' identities rather than from a PKI, with a trusted *private key generator* (PKG) computing the corresponding private keys. This alternative to traditional public-key cryptography raises specific modeling challenges for formal verification, notably the need to capture the PKG's role and the key escrow it entails. We established in [*BHS19*] the first symbolic models of identity-based cryptographic primitives in Tamarin: an abstract, parametric model covering signatures, encryption, and their combination, as well as a more precise model based on bilinear pairings that captures finer cryptographic details potentially exploitable by an adversary. We applied both models to analyze industry-proposed identity-based Authentication and Key Agreement (Authentication and Key Agreement (AKA)) protocols,[10] discovered numerous weaknesses using Tamarin, and proved the security of the corrected protocols in our abstract model.

[*BHS19*]: Basin, Hirschi, and Sasse, *Symbolic Analysis of Identity-Based Protocols*, Meadows Festschrift'19

10: Those protocols were provided to us by a major telecommunications company in the context of a collaborative research project at ETH Zurich.

3.5.2. Modeling Distributed Cryptographic Protocols

Besides protocol message and cryptographic modeling, the architecture of distributed protocols can also be challenging to model and verify. This is particularly true for Byzantine-Fault Tolerant (BFT) protocols where multiple nodes interact in a peer-to-peer fashion for which security goals are conditioned on a threshold of corrupted nodes. We have explored such modeling challenges in different contexts: the distributed Bulletin Board (BB) protocols [*HSB21*] and in blockchain protocols such as the ZCash decentralized anonymous payment protocol.

The Bulletin Board in E-Voting Protocols [*HSB21*] Physical bulletin boards are used to publish announcements, for example at the town hall. An *electronic bulletin board*, henceforth referred to as *BB*, has a similar purpose, but can be accessed *remotely*, *e.g.,* by publishing its content on a website. While BBs are deployed in various contexts, they are particularly important for electronic voting (e-voting) protocols. For e-voting to be trustworthy, the participants must be convinced that the election result is correctly computed from all eligible voters' ballots. To this end, the participants must be able to *verify* that all e-voting authorities behaved as specified, even when some of them are not trustworthy. We have extensively studied in [*HSB21*] those BBs that are used for verifiability checks, where readers read their content and check the data they read. We found out most prior e-voting protocols have a single-point-of-trust issue in that a malicious voting server can perform *equivocation attacks* and show different readers different contents on the BB, breaking verifiability and even vote privacy. Section 8.6 will provide more details about these weaknesses and attacks.

[*HSB21*]: Hirschi, Schmid, and Basin, *Fixing the Achilles Heel of E-Voting: The Bulletin Board*, CSF'21

To fix these issues, we have proposed a BB protocol that distribute trust among multiple BB *peers*, in the spirit of BFT protocols. Proving that our BB protocol satisfies the required security properties is challenging (see Remark 3.5.1). Therefore, state-of-the-art symbolic verifiers for security protocols are ill-suited for this task. Instead, we formalize in TLA+ our BB protocol as an event-based model. TLA+ has a rich specification language [CM06] based on Temporal Logic of Actions (TLA) and Zermelo-Fraenkel set theory with choice (ZFC) that we use to encode our specifications. Using the embedded proof system TLAPS [Cou+12], we prove that our protocol satisfies its security goals. We believe that our formalization is of independent interest as it provides a general framework to model and analyze distributed and cryptographic protocols, for which symbolic verification is ill-suited as explained next.

Remark 3.5.1 (On the difficulty of modeling distributed, security protocols with symbolic verifiers) A natural question that arises is whether symbolic verifiers such as Tamarin or ProVerif could be used to analyze secure, distributed protocols such as BB protocols. Indeed, we would then be able to benefit from proof automation and the ability to model cryptographic primitives in a more idiomatic manner than with TLA+, which was not designed for cryptographic protocols and requires ad-hoc encodings. However, this raises several challenges. To model distributed protocols, *e.g.,* consensus protocols, where security guarantees are parametric on the ratio of honest peers, the formalism must support describing:

(i) parameterized distributed algorithms; parameters are the number of agents of each role and a threshold δ of honest agents required for security,
(ii) cryptographic operations (*e.g.,* digital signatures for the BB),

(iii) security properties that depend on the parameters.

We realized that verifiers in the symbolic model for security protocols are unable to deal with (i) and (iii). Standard model-checkers for classical distributed programs do not work well in presence of cryptographic primitives (ii). These are the reasons why we turned to interactive proof systems and worked out our security proofs through successive refinements over a formal action systems.
We first prove a refinement that replaces cryptographic operations by their expected properties using assumptions mimicking cryptographic assumptions made in symbolic verifiers. Now that this is out of the way, we can focus on the distributed aspects. We then use classical proof techniques based on refinements in event-based models to prove that the distributed protocol satisfies the desired properties; details are given in [*HSB21*]. ▷

Remark 3.5.2 (On modeling blockchain protocols) In unpublished work, we used proverif to analyze ZCash's Sapling protocol [Hop+16], a decentralized anonymous payment scheme based on ZeroCash [Ben+14] relying on zero-knowledge proofs. Since the blockchain infrastructure cannot be faithfully modeled in symbolic verifiers (Remark 3.5.1), we idealize it as a single shared database. We model participant interactions via attacker-controlled *oracles* for triggering honest actions, writing adversarial data —subject to protocol *restrictions* modeling validation checks— and tracking operations with events to express security properties. This oracle-based technique is a reusable approach for modeling complex distributed systems in symbolic verifiers. Our analysis revealed insights on unclear hashing specifications and potentially missing hash function assumptions but no other weakness. ▷

4. Improving Verification Tools

We now summarize our contributions to improving verification tools and algorithms. We first discuss our contributions to extending the expressiveness of symbolic verifiers to handle equational theories such as XOR or purely associative symbols. We then mention on our contributions to improving the verification of equivalence properties, which are essential to model privacy properties, both for the setting of a bounded number of sessions and of an unbounded number of sessions.

4.1 Equational Theories . . 52
4.1.1 Modeling Exclusive-Or in Tamarin 52
4.1.2 Modeling Purely Associative Symbols in ProVerif and Tamarin 54
4.2 Equivalence Verification 55
4.2.1 Partial Order Reduction Techniques for the Bounded Number of Sessions Setting 55
4.2.2 Provable Sufficient Conditions for the Unbounded Number of Sessions Setting 56

4.1. Equational Theories

4.1.1. Modeling Exclusive-Or in Tamarin [*Dre+18*; *Dre+20*]

We have extended the theory behind the state-of-the-art verifier Tamarin to handle cryptographic protocols using exclusive-or [*Dre+18*; *Dre+20*]. This extension has been fully implemented in the tool and evaluated on numerous case studies (RFID protocols, e-cash protocols) showing that a large, new class of protocols, can now be analyzed. For the first time, it was then possible to faithfully analyze stateful, cryptographic protocols using XOR in the symbolic model. In particular, this extension has made possible our analyses of 5G authentication from [*Bor+19*; *Bas+18*], discussed in Chapter 7. A number of independent other analyses were also conducted thanks to this extension, such as [BST21b; BST21a; Bas+24; WZX21; NSB21; DDW22; CD19].

[*Dre+18*]: Dreier, Hirschi, Radomirovic, et al., *Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR*, CSF'18
[*Dre+20*]: Dreier, Hirschi, Radomirovic, et al., *Verification of stateful cryptographic protocols with exclusive OR*, JCS'20

Challenges The difficulty in faithfully modeling XOR is due to the unique combination of the algebraic properties of XOR, specifically:

- the associativity (A) and commutativity (C) properties and
- the cancellation properties (ca1,ca2, and ca3).

Modeling XOR without one of the two classes of properties provides only weak security guarantees since whole classes of attacks are missed.

Equational theory for XOR:

(ca1) $x \oplus x = 0$
(ca2) $x \oplus 0 = x$
(ca3) $x \oplus x \oplus y = y$
(A) $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
(C) $x \oplus y = y \oplus x$

Although Tamarin supports a large class of user-defined equational theories, one cannot simply model XOR using a user-defined theory due to its associativity and commutativity (AC) properties. Conversely, while Tamarin supports AC symbols in some built-in theories such as Diffie-Hellman exponentiation, the cancellation properties of XOR (ca1,ca2,ca3) require a special treatment, as a naive implementation results in almost systematic non-termination of the constraint solving procedure underlying Tamarin's backward search (see Section 2.4.1).

Key Ideas We model XOR in Tamarin's term algebra by introducing a new built-in operator $\cdot\oplus\cdot$ for XOR, treated modulo AC, that satisfies the expected cancellation properties[Rules]. To avoid systematic non-termination issues in Tamarin's backward search, the exploration needs to be constrained without affecting completeness (*i.e.,* preserving all traces, hence attacks). Indeed, the existing normal form conditions (see Section 2.4.1.2) are most of the time insufficient to handle XOR, as illustrated by the

Rules: **(Deduction rules for XOR)** This gives rise to the following constructor rule:

$$\frac{\mathsf{K}^{\uparrow}(x_1)\ \ \mathsf{K}^{\uparrow}(x_2)}{\mathsf{K}^{\uparrow}(x_1 \oplus x_2)}$$

and deconstructor rules:

$$\frac{\mathsf{K}^{\downarrow}(x_1 \oplus x_2)\ \ \mathsf{K}^{\uparrow}(x_2 \oplus x_3)}{\mathsf{K}^{\downarrow}(x_1 \oplus x_3)} \qquad \frac{\mathsf{K}^{\downarrow}(x_1 \oplus x_2)\ \ \mathsf{K}^{\uparrow}(x_2)}{\mathsf{K}^{\downarrow}(x_1)}$$

following example.

Example 4.1.1 Tamarin uses backwards constraint-solving to find normal dependency graphs representing the protocol executions. Suppose that during the constraint solving the adversary needs to compute a term $a \oplus b$, or more precisely Tamarin encounters an unsolved $\mathsf{K}^{\downarrow}(a \oplus b)$ premise. Then Tamarin will check all possible ways for the adversary to compute this term: The premise can either be the result of a protocol output, or the result of a deconstruction rule for XOR. In the latter case, $\mathsf{K}^{\downarrow}(a \oplus b)$ can, for example, be the conclusion of a rule instance with premises $\mathsf{K}^{\downarrow}(a \oplus c)$ and $\mathsf{K}^{\uparrow}(c \oplus b)$. Tamarin will then try to resolve the new premises, and again $\mathsf{K}^{\downarrow}(a \oplus c)$ can be the conclusion of a deconstruction rule with premises $\mathsf{K}^{\downarrow}(a \oplus d)$ and $\mathsf{K}^{\uparrow}(d \oplus c)$, and so on, resulting in non-termination. This is illustrated in Figure 4.1. Note that this is not prevented by any of the previous normal form conditions. ◊

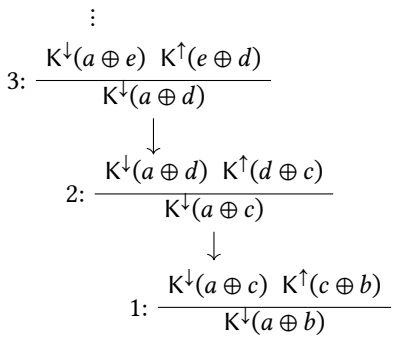


Figure 4.1.: Example 4.1.1: Infinite chain of deconstruction rules.

Such loops would occur in many cases when analyzing even simple protocols containing XOR. To prevent them, we introduce a new normal form condition $\mathrm{N}_{\oplus}$.

Definition 4.1.1 (Normal form condition $\mathbf{N}_{\oplus}$ for $\oplus$) *There is no chain of repeated instantiations of the deconstruction rules for XOR.* ◁

We can now mention the main correctness theorem, that extends Theorem 2.4.3, that states that the observable traces of the protocol are identical with the observable traces of the normal dependency graphs. This is sufficient to show soundness and completeness, as Tamarin generates all possible normal dependency graphs using constraint solving. Intuitively, $\mathrm{N}_{\oplus}$ does not limit the deducible terms for the adversary, as one can always cancel out all terms in a single step (*i.e.,* using one deconstruction rule). This is formally stated and proven below.

Theorem 4.1.1 (Extending Theorem 2.4.3, see [*Dre+20*] for a formal statement and proof) *The set of traces obtained by all executions of all dependency graphs is the same as all the traces obtained by all executions of all dependency graphs in normal forms (including the new normal form condition $\mathbf{N}_{\oplus}$ for XOR).*

Note that by relying on the observable traces we hide the adversary's deduction steps on both sides (which slightly differ due to the normalized deduction), but ensure that security properties (defined on action facts) are carried over correctly. This theorem shows that by strictly ordering the K-facts to comply with the normal form condition, the adversary does not lose any power. Finally, we update the constraint solving procedure (Section 2.4.1.3) to enforce this new normal form condition at the level of constraint systems, and we formally prove the procedure is still sound and complete by extending [Mei13, Theorem 4].

Moreover, we show that the previous attacker model of Tamarin [BDS15] is not suitable for XOR when analyzing equivalence properties[1] as it yields spurious attacks (*i.e.,* attacks that are solely artifacts of this internal model). We improve the attacker model to considerably enhance the precision of the analysis in the presence of XOR, we leave the details to [*Dre+20*].

We have implemented these extensions in Tamarin and demonstrated that the tool succeeds to effectively analyze diverse protocols including some that were previously out of scope of automated verification (RFID protocols and digital cash protocols). Later, we heavily leveraged this extension to analyze the 5G authentication protocol and find new attacks on it [*Bor+19*; *Bas+18*].

1: We have later discovered that the existing algorithm for verifying equivalence properties in Tamarin [BDS15] was and is still unsound for some specific uses of AC symbols, including XOR. Essentially, these cases may occur when the differing terms in the two sides of the equivalence may have different variants or AC-unifiers when performing the folding variant narrowing procedure [ESM12]. Since then, we have been working, with most of the Tamarin core team members, on a complete redesign of this algorithm, which is still ongoing work. Note that this is strictly limited to the verification of equivalence properties and does not affect the verification of reachability properties. Another work [DKK26] has identified and fixed an independent issue in the interaction of an underspecified feature of the implementation of the constraint solving procedure and the use of XOR, which also caused soundness issues. This is now fixed. We stress that both issues did not contradict our theorems [*Dre+18*; *Dre+20*] but reveal flaws in earlier works we built upon.

4.1.2. Modeling Purely Associative Symbols in ProVerif and Tamarin [*Che+23*]

As explained in Section 3.3, modeling weak hash functions in security protocols requires to faithfully model the associative concatenation operator $\|$. However, at the time, none of the existing protocol verification tools supported this, especially the state-of-the-art tools Tamarin and ProVerif. We added support for purely associative symbols to both tools [*Che+23*].

[*Che+23*]: Cheval, Cremers, Dax, et al., *Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses*, USENIX'23

Challenges The core difficulty is that both tools rely on unification. Given two messages, one needs to be able to compute a finite and complete set of most general unifiers, *i.e.,* a set of substitutions that represents all possible ways of instantiating the messages that make them equal. For instance, $\langle x, 0\rangle \stackrel{?}{=} \langle 1, y\rangle$ has a unique unifier $\{x \mapsto 1, y \mapsto 0\}$. For associative operators, the issue is that such a set is not always finite. For instance, $0\|x \stackrel{?}{=} x\|0$ has an infinite set of unifiers of the form $\{x \mapsto 0^n \mid n \in \mathbb{N}\}$.

A first approach is to model the associativity under a hash function operator for a bounded depth only and to use the built-in pairing operators $\langle\cdot,\cdot\rangle$ with extra equations to model this "bounded associativity". For instance, one could specify $h(\langle x, \langle y, z\rangle\rangle) = h(\langle\langle x, y\rangle, z\rangle)$, but this does not imply that $h(\langle x, \langle y, \langle z, z'\rangle\rangle\rangle) = h(\langle\langle\langle x, y\rangle, z\rangle, z'\rangle)$. We used such a bounded modeling successfully on some examples. However, this modeling may miss attacks that require associativity at a deeper depth than modeled. We explain next how we overcame this problem in both tools.

Key ideas We extended Tamarin and ProVerif to support associative operators such as $\|$. For Tamarin, we make the observation that the unification problems arising from models using $\|$ *can* be infinitary but that it suffices for the actual instances occuring during protocol verification to yield finite sets of unifiers for the analysis to be sound. We integrated recent advances from the Maude [Dur+18] backend that deals with such unification problems, and stops the verification when Maude is unable to compute a finite solution. The latter rarely occurred in practice and only affected a single property in one out of 21 analyses, indicating practical feasibility. Next, we introduce constructor and deconstructor rules$^{\text{Rules}}$ for $\|$ and noticed that new normal form conditions are needed to ensure termination of the backward search. We introduced for this two new normal form conditions and proved the soundness and completeness of the resulting procedure. We did so by adapting Theorem 2.4.3 and [Mei13, Theorem 4] (as we did for XOR). We leave out the details of the extensions and their soundness and completeness proofs, which are provided in [*Che+23*].

Rules: **(Deduction rules for $\|$)** New rules, respectively deconstructor and constructor rule:

$$\frac{\mathsf{K}^{\downarrow}(t_1\|\ldots\|t_n)}{\mathsf{K}^{\downarrow}(t_1)\ \ldots\ \mathsf{K}^{\downarrow}(t_n)} \qquad \frac{\mathsf{K}^{\uparrow}(t_1)\ \ldots\ \mathsf{K}^{\uparrow}(t_n)}{\mathsf{K}^{\uparrow}(t_1\|\ldots\|t_n)}$$

Definition 4.1.2 (Normal form conditions for $\|$) *(simplified version)*

- $N_{\|}^1$ *There is no construction rule for $\|$ that has a premise of the form $\mathsf{K}^{\uparrow}(s\|t)$ and all conclusion facts of the form $\mathsf{K}^{\uparrow}(s\|t)$ are conclusions of a construction rule for $\|$.*
- $N_{\|}^2$ *The conclusion of a deconstruction rule for $\|$ is never of the form $\mathsf{K}^{\downarrow}(s\|t)$.*

◁

For ProVerif, we developed and used a new extension, namely *recursive computation functions*: powerful generic user-defined functions to manipulate messages that are fully integrated in the reasoning engine. For instance, we define the computation function H that recursively computes the MD construction (removing the need for an associative $\|$ operator). The details and soundness of this extension are also provided in [*Che+23*].

4.2. Equivalence Verification

We also contributed to improving the verification of equivalence properties, essential for modeling privacy. Both main verification approaches suffer from distinct limitations: in the setting of a bounded number of sessions, the *state space explosion problem* caused by protocol concurrency; in the unbounded setting, the over-approximation of behavioral equivalence, which is too imprecise for privacy properties such as unlinkability. Building on foundational work from our PhD thesis [*Hir17*], we have later contributed new techniques [*HC19*; *BDH18*] that partially address these limitations. While theoretically grounded, our solutions have been implemented and validated on real-world protocol case studies and are supported by tool integrations.

4.2.1. Partial Order Reduction Techniques for the Bounded Number of Sessions Setting

Tools deciding behavioral equivalence for a bounded number of sessions must symbolically explore all execution traces, leading to severe *state space explosion*. For instance, prior to our contributions, verifying anonymity in the Private Authentication protocol [AF04] using APTE took under 0.1s for one session of each role, but over an hour for two and over two days for three sessions.

In standard model-checking, *Partial Order Reduction* (POR) techniques [Pel98; God96] mitigate this by exploiting commutativity of independent actions. However, as noted by Clarke *et al.* [CJM03], POR cannot be directly applied to security protocols: the active attacker can construct arbitrary messages from past outputs, making *a priori* all inputs depend on all outputs and yielding a very large dependency relation and extremely limited commutativity. Additionally, symbolic semantics complicates matters (symbolically different states may represent overlapping concrete state sets), and POR must preserve trace equivalence —not just reachability— requiring *symmetric reduction* on both sides of the equivalence.

Our Earlier Contributions During our PhD [*Hir17*], we developed dedicated POR techniques for trace equivalence checking [*BDH14*; *BDH15*; *BDH17*], restricted to *action-determinate protocols* (*i.e.,* where the system is deterministic once an observable trace is fixed).[2] A key challenge was compatibility with symbolic execution: the reduction must work when messages remain unknown yet leverage information as inferred by the constraint solver. Our approach refines the interleaving semantics in two steps: *compression* (a data-independent strategy based on process polarity) and *reduction* (a data-aware, optimal elimination of redundant traces). We proved soundness for action-determinate protocols [*BDH15*], integrated the techniques into APTE [*BDH17*] with dramatic speedups (*e.g.,* 7 sessions of Private Authentication now tractable), and they have since been adopted by all state-of-the-art tools in this setting: APTE, DeepSec [CKR18], Spec [TD10], AKISS [Cha+16].

2: Trace equivalence is one form of behavioral equivalence (see Section 2.3.3), where two systems are equivalent if they have the same set of observable traces.

New POR Technique: Beyond Action-Determinism The above POR techniques crucially rely on action-determinism, excluding many interesting protocols and properties —notably, there is no precise modeling of unlinkability within this class. We completely lifted this restriction [*BDH18*]

by designing POR techniques that are sound for trace equivalence without any assumption on the protocols, leveraging *stubborn sets* and *sleep sets* from classical POR [God96] while introducing a novel symbolic form of independence computations to handle the specificities of symbolic execution. We implemented a standalone library (Porridge [3]) and integrated it into both APTE [Che14] and DeepSec [CKR18], with experimental results confirming its effectiveness, though with smaller speed-ups than under action-determinism.

3: https://members.loria.fr/LHirschi/porridge/index.html

4.2.2. Provable Sufficient Conditions for the Unbounded Number of Sessions Setting

We now turn to tools that semi-decide trace equivalence for an unbounded number of sessions (*e.g.,* ProVerif and Tamarin). These tools all rely on *diff-equivalence*, a strong over-approximation of trace equivalence that requires the two protocols to share the same structure and execute in lockstep —including internal rules (*e.g.,* conditional esubstitutions) invisible to the attacker. Despite partial improvements [CB13b], the precision gap remains significant in general.

While this gap is harmless for some properties (*e.g.,* strong-secrecy), it is problematic for privacy goals such as *unlinkability* or *vote privacy*. For instance, the formal definition of strong unlinkability [Ara+10] requires verifying an equivalence between protocols that inherently do not share the same structure, causing diff-equivalence to systematically report false attacks. In practice, tools like ProVerif and Tamarin cannot be used off-the-shelf for unlinkability of many protocols, *e.g.,* RFID and e-passports protocols such as BAC, PACE, Feldhofer, Hash-Lock, LAK, *etc.*

Our Earlier Works on Unlinkability and Anonymity for 2-agents Protocols Rather than improving diff-equivalence to get closer to trace equivalence [CB13b; BS16], we took a different approach [*HBD16*; *HBD19*]: for a class of protocols and privacy goals, we identified *sufficient conditions* that are much simpler to verify automatically.

For unlinkability and anonymity of 2-party protocols, we defined two conditions —one preventing control-flow leaks, the other data leaks— and proved they are always sufficient to guarantee both properties. Crucially, these conditions fall within the scope of existing tools like ProVerif without the systematic precision issues of diff-equivalence. We implemented this as UKano [4], which automatically generates and checks ProVerif models for both conditions. The method proved precise enough to establish the first proofs of unlinkability for several protocols (ABCDH, BAC+PA+AA in e-passports), uncovered a weakness in PACE and a new traceability attack on LAK (previously claimed untraceable), and enabled first automatic unlinkability analyses for PACE, Feldhofer, Hash-Lock, and LAK. Our conditions turned out to be tight in practice: every condition violation that was found corresponded to a concrete attack. This technique has since been independently extended to stateful protocols [BDM20].

4: https://projects.lsv.ens-paris-saclay.fr/ukano/

Vote Privacy for E-Voting Protocols To demonstrate the generality of our sufficient-conditions methodology, we applied it [*HC19*] to vote privacy for e-voting protocols, defining three conditions that together guarantee vote privacy for a large class of protocols. Compared to existing automated approaches, our method substantially expands the analyzable class of protocols and threat models —handling honest authorities across

phases, threat models without dishonest voters, and protocols whose privacy depends on fresh data from other phases— and significantly improves verification efficiency (*e.g.*, over two orders of magnitude speedup for the LEE protocol). We formally prove the soundness of our method: if the three conditions hold, then vote privacy is guaranteed. We validated our approach using ProVerif on FOO, LEE, JCJ, and Belenios, with no false attacks, suggesting tight conditions.

5. New Proof Methodologies

🔍 *Focus on Lattice Exploration [Gir+20; Che+23; DHK25a]*

Having a protocol model and some security properties that are in scope of existing verification tools is obviously the first step towards a formal analysis. However, in practice, the verification tools for the unbounded setting (*e.g.,* ProVerif, Tamarin) often require some manual guidance to effectively perform the analysis as false attacks or non-termination may otherwise occur. We have developed a proof methodology in ProVerif to guide the proof search that we claim is of general interest, we briefly present it in Section 5.6.

Assuming the verification successfully terminates, another practical challenge is to ensure that the analysis covers the relevant threat models. We have observed, as well as others [BC14], that automated formal verification in the symbolic model is particularly well-suited to systematically explore a large space of threat models and security properties. This has led us to transition from a single verification of some expected properties to a systematic exploration of a *lattice* of threat models and properties producing fine-grained results that can be analyzed to gain insights on the protocol. In a sense, this verification methodology enables *measuring* the level of security of a protocol by discovering which security properties hold and when: To which (combinations of) threats is it resilient, and to which not? What level of confidentiality and authentication is guaranteed to hold? What are the trade-offs between strong security properties and stronger threat models (*e.g.,* injective agreement versus non-injective agreement with some additional key compromise)? Crucially, this captures and goes beyond the known classes of attacks and threat models such as: Key Compromise Impersonation (KCI), Uknown Key-Share (UKS), Harvest-Now-Decrypt-Later, Perfect Forward Secrecy (PFS), Post-Compromise Security (PCS), etc. This is particularly important as we all witness ever-increasingly stronger security and privacy expectations for security protocols. For example, PFS and PCS are now standard requirements for messaging protocols and are likely to be for key exchange protocols in the near future.

Beyond *measuring security*, another advantage of this methodology is that it enables the security comparison of different protocols, which was key in our Noise analysis [*Gir+20*], see Sections 3.4 and 5.5. Moreover, it also allows exploring variants and features of the protocol under study, better understand their impact on security and on the verification challenges (calling for specific proof heuristics as we did for OPC UA, see Section 5.1.3). Finally, it also enables identifying minimal assumptions and protocol features required for an attack, yielding a form of "minimal attack scenario". Hence the following question:

> *How to* effectively measure and compare the security *of cryptographic protocols through DY formal verification?*

The challenges that come with such an analysis approach are twofold. First, it yields a large number of verification goals, which may be impractical to verify one by one. Second, even assuming all verification goals can be answered, it is not clear how to analyze and interpret the results

5.1 Dealing with a Large Collection of Verification Goals 59
5.1.1 Noise Framework: Systematically Compare Protocols under Various Threat Models 59
5.1.2 Hash Weaknesses: Measuring Resistance to Various Threats and Weaknesses . 60
5.1.3 OPC UA: Taming Non-Termination With Protocol Feature-Specific Proof Heuristics 60
5.2 Defining a Lattice of Verification Goals 61
5.3 Static Analysis 64
5.4 Lattice Exploration . . . 65
5.5 Example of Results . . . 67
5.6 Proof Helper Methodology for ProVerif 68

in a meaningful way; what one can do with 10^{12} proof/attack results? We present a methodology addressing these challenges in the remaining of this Chapter, focusing on our case study of the Noise analysis [*Gir+20*]. Note that we have iteratively reworked and improved this methodology over different works [*Gir+20*; *Che+23*; *DHK25a*], we thus present here the most recent and advanced version of it. Note also a recent thesis [Mah25] that builds on our prior works and [BC14] to propose a generic definition of the lattice of verification goals, *i.e.,* individual queries that must be proved by a protocol verifier such as Tamarin or ProVerif, although it does not focus on efficient lattice exploration techniques.

5.1. Dealing with a Large Collection of Verification Goals

We encountered various situations where fine-grained and meaningful results required to consider a large collection of verification goals, which motivate the methodology we present next. The latter is mostly verifier-agnostic, and we applied it in different contexts and with different tools (ProVerif and Tamarin), as we shall see in the next sections.

5.1.1. Noise Framework: Systematically Compare Protocols under Various Threat Models [*Gir+20*]

As explained in Section 2.2.2, the Noise framework allows one to specify a very large, albeit finite set of different protocols. The specification defines 53 pre-defined (two ways) patterns but at least hundreds others could also be specified in this language. Moreover, for each of these pattern and corresponding protocol, many different security properties should be analyzed. Indeed, recall from Section 2.2.2 that the security of Noise messages depend on the flight from which the message was sent, *e.g.,* agreement and secrecy properties are stronger for the second flight than for the first flight.

On top of that, there exist various levels of security for each of the different security properties depending on the underlying threat model. As an illustration and continuing Example 2.3.9, some levels assume the recipient's private key has been compromised and others assume that the sender's static key has been compromised. For this reason, we considered 13 different compromises (*e.g.,* sender's static key, recipient's ephemeral key) and arbitrary combinations of such atomic adversarial capabilities.[1] This combination of different protocol variants, security properties to check under many different threat models yield a huge and impractical number of verification goals[2].

As we shall see next, organizing (i) verification goals in a lattice and (ii) this large proof effort into an optimized lattice exploration enabled us [*Gir+20*] to answer the following two questions:

> *First, which Noise protocol should practitioners use for a given scenario and initial key distribution? Second, Noise theoretically offers an unbounded number of protocols, but are they all interesting, or are some Noise protocols subsumed by others?*

1: These capabilities will be presented in more details in Section 5.2 and Figure 5.1.

2: 410 000 for the 53 pre-defined Noise patterns.

5.1.2. Hash Weaknesses: Measuring Resistance to Various Threats and Weaknesses [*Che+23*]

Protocol languages such as the Noise framework are not the only situation where a large number of verification goals naturally arise. As just mentioned, this happens when considering various threat model dimensions that can be independently combined, yielding a combinatorial explosion. This can stem from analyzes which aim to *measure* the level of security instead of evaluating a few pre-defined security properties and threat models. Beyond this case, there are cases where we simply do not know in advance the expected guarantees, in which case we are left with a property and threat model space we shall explore. A good illustration of this situation is our analysis of hash function weaknesses in security protocols [*Che+23*] which precisely aim to answer which protocols resist which hash weaknesses. Our OPC UA analysis, discussed below, also falls into this category.

Recall from Section 3.3 that we considered a lattice of threat models reflecting various adversarial capabilities to exploit hash function weaknesses (see Figure 3.2). In practice, this yields 264 distinct scenarios to consider for each protocol model and security property to verify. As an illustration, our esubstitution on 4 protocols with standard confidentiality and authentication properties yielded more than 5000 verification goals to verify in total.

5.1.3. OPC UA: Taming Non-Termination With Protocol Feature-Specific Proof Heuristics [*DHK25a*]

Finally, even when considering a few threat models and security goals, hence yielding a manageable number of verification goals in theory (here only 5 verification goals), we sometimes need to split the verification effort in order to design and apply goal-specific proof heuristics. We had to do so for our analysis of the OPC UA protocol [*DHK25a*] with ProVerif whose results will be presented in Chapter 6.

As we shall see, OPC UA is a complex protocol with many features that make its automated analysis challenging. This translates into non-termination or false attacks when trying to verify any of the 5 verification goals with a single model and standard proof strategy. Using ProVerif out of the box was simply not sufficient to obtain any meaningful result.

In order to isolate the different model features that were challenging for ProVerif to analyze, we developed a modular protocol model that allows us to choose: *(i)* a particular protocol configuration, *(ii)* the possibility for compromise of long-term keys and/or channel keys, or no key leakage at all, *(iii)* to turn on / off some protocol features (*e.g.,* channel reopening and session switching). As each of these model options are pairwise independent, we end up with 7 056 possible configurations and as many models.

While we ultimately aim to provide results for the *maximal* configuration, *i.e.,* the one with all features, threat capabilities, and richest configuration, we had to consider simpler configurations first in order to identify and design proof heuristics that would then be applied to the maximal configuration. Moreover, first exploring simpler configurations allows minimizing attack traces and narrow down the exact protocol features that are required to trigger a vulnerability.

5.2. Defining a Lattice of Verification Goals

The first step to tame large-scale verification efforts is to organize the verification goals into a lattice structure. Each node corresponds to a specific verification goal to verify. The nodes are partially ordered by a "trivial implication" relation we statically identify; *e.g.,* a property remains true when reducing adversarial capabilities. We informally illustrate this idea on our analysis of the Noise framework, a more formal framework can be found in [Gir19; *Gir+20*].

Verification Goals When analyzing the Noise family of protocols, we had to verify each of the security property with respect to a range of threat models, which are modeled by a set of adversary's capabilities. A security property and a threat model together specify a *verification goal* (or *proof obligation*), which we can evaluate with a protocol analysis tool such as ProVerif or Tamarin.

To motivate our formulation, let us focus first on secrecy and agreement properties. These are first-order logic formula over trace executions of the form:

$$P := \tau \implies C,$$

where τ represents a *trigger* that occurs whenever the security property P in question applies (*e.g.,* upon reception of a message for agreement properties) and C describes the *claim* expressing the guarantees expected to hold for that property (*e.g.,* secrecy of the exchanged payload for a secrecy property). For our analysis of Noise, we considered the following properties: (i) secrecy of a given payload from the perspective of a given role, (ii) non-injective and injective agreements of the session until a given flight from the perspective of a given role, (iii) anonymity of a given role with respect to its public static key.

In a model where all attacker capabilities are enabled, such a property P is not expected to hold in general. We thus refine a property P into a *verification goal* P_t of the form:

$$P_t := \tau \implies (C \vee t),$$

where t describes a threat model, *i.e.,* a combination of adversarial capabilities. (See an example on the side and a detailed example for Noise in Example 5.2.1.) Thus, a verification goal P_t is a statement that the protocol provides the guarantees of the property P we consider, unless the adversary has access to the capabilities described in t.

Continuing Example 2.3.7, the authentication property

$$\forall \mathsf{pk}_A, \mathsf{pk}_B, i.\ \mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)@i \implies \exists j.\ \mathsf{Start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)@j$$

is of this form with $\mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ as the trigger τ and $\mathsf{Start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)$ as the claim C.

We assume the model is extended to support dynamic compromise of the agents' private keys: sk_X can be compromised at any time, in which case the action fact $\mathsf{Compr}(\mathsf{pk}_X)$ is fired. The above authentication property is then refined into the following verification goal:

$$\forall \mathsf{pk}_A, \mathsf{pk}_B, i.\ \mathsf{End}_\mathsf{A}(\mathsf{pk}_A, \mathsf{pk}_B, k)@i \implies \exists j, l.\ (\mathsf{Start}_\mathsf{B}(\mathsf{pk}_A, \mathsf{pk}_B, k)@j \vee \mathsf{Compr}(\mathsf{pk}_B)@l)$$

This goal models that the authentication property holds unless the adversary compromises the peer's private key, in which case a trivial impersonation attack is possible. This way, we notably capture KCI attacks, in that authentication must be guaranteed even if the recipient's private key is compromised.

Threat Models Intuitively, each threat model t can be expressed as a combination of atomic *adversarial capabilities*. We summarize these capabilities we considered for Noise protocols in Figure 5.1, deferring a more formal interpretation to [*Gir+20*, §3]. We now model an adversary who possesses a given subset of these capabilities, including all or none of them. We first describe how these capabilities can be combined into a threat model and afterward how they can be used to evaluate a security property.

Definition 5.2.1 *Let A be the set of adversary capabilities given in Figure 5.1. We define the set of* threat models*, denoted by T, to be the (subset of) propositional logic formulas, built from A, $\wedge$, $\vee$, and the bottom element $\bot$, which represents an empty threat model.* ◁

$R_{\mathtt{e}}$	Actor ephemeral key is revealed
$R_{\mathtt{re}}$	Peer's ephemeral key is revealed
$R_{\mathtt{s}}$	Actor's static key is revealed
$R_{\mathtt{rs}}$	Peer's static key is revealed
$R_{\mathtt{psk}}$	The pre-shared key owned by the actor for this session is revealed
$R^{<}_{\mathtt{e}}$	Actor's ephemeral key is revealed before the claim
$R^{<}_{\mathtt{re}}$	Peer's ephemeral key is revealed before the claim
$R^{<}_{\mathtt{s}}$	Actor's static key is revealed before the claim
$R^{<}_{\mathtt{rs}}$	Peer's static key is revealed before the claim
$R^{<}_{\mathtt{psk}}$	The pre-shared key owned by the actor for this session is revealed before the claim
$D_{\mathtt{pki}}$	Dishonest pre-message PKI
$D_{\mathtt{re}}$	Peer's ephemeral key is dishonestly generated
$D_{\mathtt{rs}}$	Peer's static key is dishonestly generated
active	Active adversary

Figure 5.1.: Atomic adversary capabilities; the set thereof is denoted A.

Note that, when combined with a security property P to form a verification goal P_t, the threat model t plays the role of an *escape clause*: t characterizes the adversarial behaviors under which the claim is *not* required to hold. A larger t thus yields a weaker verification goal. In particular, the empty threat model $\bot$ admits no escape and so requires the claim to hold in all circumstances ($P = P_\bot$), whereas the maximal threat model yields the weakest verification goal. However, not all combinations of capabilities are meaningful. For example, $R^{<}_{\mathtt{rs}} \wedge R_{\mathtt{rs}}$ is intuitively equivalent to the threat model $R^{<}_{\mathtt{rs}}$ as revealing the key prior to the claim also satisfies the requirement to reveal the key at any point. We define a notion of redundancy that we use to eliminate such redundant threat models.

Definition 5.2.2 *We define $\preceq$ to be the smallest reflexive and transitive relation[3] over threat models containing: $R^{<}_x \preceq R_x$ for $x \in \{\mathtt{e}, \mathtt{re}, \mathtt{s}, \mathtt{rs}, \mathtt{psk}\}$, $D_{\mathtt{pki}} \preceq D_{\mathtt{rs}}$, and $D_x \preceq$ active, for $x \in \{\mathtt{rs}, \mathtt{re}, \mathtt{pki}\}$. For $t_1, t_2 \in T$, we say that t_1* subsumes *t_2 when $t_1 \preceq t_2$.* ◁

3: This relation intuitively captures the idea that one threat model is more general than another, in the sense that it allows for more adversarial capabilities. For example, all adversarial behaviors possible under D_x are also possible under an active attacker, hence $D_x \preceq$ active.

We can use this to reduce the number of relevant threat models using the following result [*Gir+20*].

Theorem 5.2.1 ([*Gir+20*]) *Let P be a property, and $t_1, t_2 \in T$ be such that $t_1 \preceq t_2$. If P_{t_1} holds then P_{t_2} holds as well.*

Our ordering $\preceq$ induces a partial order on the set of threat models T, and thus yields an equivalence relation $\simeq$ defined as $t_1 \simeq t_2$ when $t_1 \preceq t_2$ and $t_2 \preceq t_1$. We denote as $\bar{T}$ the quotient of the set of threat models by $\simeq$. The set $\bar{T}$ represents the set of *distinct threat models* that we will consider.

$\bar{T}$ is still large: it contains more than 10^{12} elements. Although this indicates how fine-grained our analysis is, this large number poses two problems. First, the raw results of evaluating these threat models against each claim would be beyond human comprehension. We show next how to condense these results into a single summary statement without any loss of precision. Second, evaluating all of these threat models would take substantial computational resources. We address these problems with optimized lattice exploration strategies (Sections 5.3 and 5.4).

Strongest Threat Model We now show that for a given property, there exists a unique element of $\bar{T}$ that subsumes exactly those threat models under which the property holds. This allows us to summarize succinctly

the conditions under which a property holds.

Theorem 5.2.2 (Strongest Threat Model (STM) [*Gir+20*; Gir19]) *Let P be a property. Let $\bar{T}(P)$ be the set of distinct threat models $t \in \bar{T}$ under which P_t is true. There exists a unique element in $\bar{T}(P)$ that subsumes all other threat models in $\bar{T}(P)$. We denote this element by $B(P)$ and call it the* STM *for P.*

Without loss of generality, we can represent the unique STM by a representative of $B(P)$ in Disjunctive Normal Form (DNF), that is, as a disjunction of clauses, where each clause is a conjunction ($\wedge$) of adversarial capabilities. Each clause corresponds directly to a minimal set of capabilities required for the adversary to violate the security property. This makes it easy to interpret the STM as a set of minimal attack scenarios. Informally we will refer to this representative in DNF as *the STM*.

Example 5.2.1 Secrecy of the third payload of I1N from the perspective of the initiator (called claimer) holds under the strongest threat model: $R_{\mathrm{re}} \vee D_{\mathrm{re}} \vee (R_{\mathrm{e}} \wedge R_{\mathrm{s}})$. This is equivalent to the following statement, where p denotes the payload:

$$Trigger(p) \implies Secret(p) \vee (R_{\mathrm{re}} \vee D_{\mathrm{re}} \vee (R_{\mathrm{e}} \wedge R_{\mathrm{s}}))$$

Which means *one* of the following must be true:

- The secrecy claim on the payload p holds, *i.e.,* $Secret(p)$.
- The adversary compromised the peer's ephemeral key.
- The peer's ephemeral key was generated by the adversary.
- The adversary compromised both the claimer's ephemeral and static key (as modeled in the conjunct $R_{\mathrm{e}} \wedge R_{\mathrm{s}}$).

These cases cover all possible attacks using combinations of atomic adversarial capabilities. Furthermore, each case is minimal, *e.g.,* in the fourth case it must be that no attack is possible if the adversary only compromises the claimer's ephemeral key (R_{e}) but not the claimer's static key (R_{s}). ◊

Consequently, for each property P (secrecy, non-injective agreement, injective agreement, and anonymity), we can condense the result to a single threat model $B(P)$, which summarizes the exact capabilities the adversary needs to violate the property given by P. This reduces our set of results for all properties to where they can be inspected by hand. However, we must still compute this STM. Naively, a brute force strategy enumerating all threat models in $\bar{T}$ would suffice, where we submit all verification goals as lemmas to Tamarin for each property. However, this would yield more than 10^{12} verification goals per handshake, message, and security property. We explain how to tame this complexity next.

We first exemplify how this choice of threat models, combined with security properties, is expressive enough to encode well-known standard security notions.

Example 5.2.2 Secrecy under $R^{<}_{\mathrm{rs}} \vee R^{<}_{\mathrm{s}}$ captures a form of PFS where payload secrecy holds **unless** the actor's or the actor's peer's static, private key is compromised before the claim. ◊

Example 5.2.3 (KCI resistance) KCI resistance can be modeled as injective agreement under the threat model $R_e \vee R_{\mathrm{re}} \vee D_{\mathrm{re}} \vee R_{\mathrm{rs}} \vee D_{\mathrm{rs}}$. In plain English, agreement holds unless the actor's ephemeral key is compromised, or an asymmetric key of the actor's peer is either compromised or was in fact generated by the adversary. Hence even if the actor's static key is compromised, agreement still holds. ◊

Lattice of Verification Goals We lift our definition of subsumption (Definition 3) from threat models to verification goals in the natural way. Namely, the subsumption relation is the smallest reflexive, transitive relation such that:

- if t subsumes t' ($t \preceq t'$), then for any property P, P_t subsumes $P_{t'}$ ($P_t \preceq P_{t'}$);
- for any given threat model t, message, and set of keys, injective agreement goals subsume non-injective agreement goals; and
- for any given threat model and message, if S and S' are sets of keys where $S \subseteq S'$, then (non-)injective agreement on this message and keys S' subsumes (non-)injective agreement on the same message and keys S.

The resulting subsumption over-approximates, but does not coincide with, entailment, *i.e.,* for verification goals P_1, P_2, if P_1 subsumes P_2 ($P_1 \preceq P_2$) and P_1 is true then P_2 must also be true.[4] Note that the converse is wrong in general: we might fail to statically recognize some entailments. Our goal is now to leverage this to efficiently *evaluate* this lattice, that is to determine which verification goals are true or false.

4: This may seem counter-intuitive at first since $t \preceq t'$ means that t corresponds to a weaker threat model than t' and yet $P_t \preceq P_{t'}$, *i.e.,* P_t is a stronger property than $P_{t'}$. This is because t acts as an escape clause in $P_t = \tau \Rightarrow (C \vee t)$: a smaller t leaves fewer escapes, hence yielding a stronger verification goal.

Other Kind of Lattices Each lattice we used for our analysis of Noise was relative to a fixed protocol model. However, we also considered in other works lattices spanning different protocol models and security properties. For example for our OPC UA analysis [*DHK25a*], different protocol modes and features (see Chapter 6) with their respective models are present in the same lattice. Yet, the partial order over the nodes can still relate these nodes: for example a (trace) property (such as authentication) that holds with a protocol feature enabled still holds when it is disabled. The notion of maximal nodes (with respect to the partial order) corresponding to verification goals that hold is still meaningful. In this case, it notably allows identifying the minimal set of protocol features that can be exploited by the adversary to violate a security property.

5.3. Static Analysis

Before even calling a protocol verifier to evaluate the verification goals from the lattice, we apply some simple static analyzes to identify (i) further nodes that are comparable (one subsume the other) or equivalent and extend the partial order accordingly, (ii) nodes for which we can statically conclude whether the property holds or not, and populate the lattice with these results. This allows us to dramatically reduce the number of calls to the protocol verifier. We summarize some of these analyses we used for our Noise analysis [*Gir+20*].

Threat Models and Handshakes Taking the quotient of T by $\simeq$ reduces the number of distinct conjunctions of atomic capabilities in A from 16,384 to 1,701. We explain and prove in [Gir19] how and why results for conjuncts are enough to compute the STMs. We can also consider how elements of A interact with the property under consideration. For example, if a property considers a point in the protocol where an ephemeral key for a party has not yet been instantiated, we need not consider this key's reveal. Similarly, where the handshake pattern has no pre-messages, $D_{\texttt{pki}}$ gives the adversary no additional power, allowing us to extend the lattice partial order accordingly.

Threat Models and Properties Next, we note that when analyzing agreement, a passive adversary cannot make use of any knowledge gained to affect the views or actions of the other participants, because they cannot insert their own messages. Hence, we need not consider key reveals whilst considering agreement for passive adversaries. Similarly, for a passive adversary attempting to violate a secrecy property, the timing of a key reveal does not change the adversary's ultimate knowledge set, which means that there is no difference between the timed and untimed variants of a reveal, and we can infer the result for one by evaluating the other. Furthermore, revealing a key after a claim does not increase the adversary's ability to violate non-injective agreement at the time of the claim.

Trivial Attacks In many threat models, the adversary may have enough knowledge to perform a *trivial attack* on a handshake. For example, if the adversary completely compromises the peer's state, then secrecy and agreement properties no longer hold. Similarly, if the adversary learns any PSKs present and at least one private key for each Diffie Hellman operation, they can compute the session key. We generalize these observations into a wider category of trivial attacks, which are important for our tool's efficiency. For such cases, we can immediately conclude that the goal is false based on a simple static analysis; and then propagate the results through the lattice as all goals that subsume such false goals are false as well.

The above observations allow us to immediately infer that over 99% of verification goals are false. This leaves us with, on average, only 63 verification goals per remaining property, as opposed to the 16,384 naive ones. There is a varying number of properties per pattern, depending on, *e.g.,* its number of messages. Overall, this leaves us with about 410,000 verification goals for 53 patterns. Note that we show in [Gir19] that all our reductions are sound in that we have formally proven that they do not impact the actual results.

Other Heuristics In other works, we also identify and exploit implications between threat models (and not only adversarial atomic capabilities as in Definition 3). For instance, recall our lattice of hash function weaknesses from Figure 3.2 where the arrows correspond to $\succeq$. This lattice contains some cross-dimensions redundancies that are not captured by the joint partial order on each dimension: we have that $allCol \succeq hashExt$, $sndPreImg \wedge inLeak \succeq fstPreImg$, and $inLeak \succeq hashExt$.

5.4. Lattice Exploration

We can now consider the verification goals from the lattice and submit them to the protocol verifier (*e.g.,* Tamarin or ProVerif), written as formal properties (as Tamarin's lemmas or ProVerif's queries), in order to label each of the node with True (goal is proven to hold) or False (an attack is found). However, we can use the subsumption relationship to speed up this labeling. For example, if a goal is true, then all weaker goals (that subsume the true goal) must also be true. Likewise, if an attack is found for a goal, then the attack also holds for any stronger goal (that are subsumed by the false goal). We still have to choose a tree traversal strategy, *i.e.,* in which order we perform the individual proof obligations and propagate the results.

Tree Traversal Strategies A first valid and easy to deploy tree traversal strategy is based on a maximal-payoff first search, *i.e.,* the payoff here proxies how many tasks we could save when propagating the result from the explored goal. We designed for this a dedicated heuristic that approximates the expected payoff [*Gir+20*]. Overall, this reduces the number of verification goals from 410,000 to 150,000 calls to Tamarin. Again, the reduction is provably sound [Gir19].

Multi-Round Traversal Strategy An inherent issue with such strategies is the unpredictable running time of the prover for analyzing one goal. Although a goal might be the best option in terms of payoff, if we must wait for unreasonably long to obtain this result, it was not the best choice after all. For this reason, we designed multi-round exploration strategies. We try to fully explore the lattice at each round but with some strict time and memory[5] budget: we kill the verification task when the prover exceeds the budget and nonetheless continue exploring. At the end of the first round that uses a low budget, and that runs reasonably fast, we have likely not obtained results for the full lattice. However, we obtained results for the "low-hanging fruits" yet allowing us to prune large parts of the lattice through subsumption. We then iteratively increase the budget for the next round and iterate like this until the lattice is fully explored. We stress that all our strategies are fully mechanized with tool support [*Gir+20*; *Che+23*; *DHK25a*]. Such an exploration of our lattices modeling hash weaknesses for 5 protocols had 5000 proof obligations and required 1600 actual calls to Tamarin, realized in 150 minutes in total.

5: RAM consumption

Parallelism Finally, we also leverage multi-threading and explore the lattice in parallel on multiple CPU cores. There is a trade-off here since the scale effort will not be linear with the number of cores. Indeed, we sometimes hit race conditions where one objective analysis result would subsume another objective analysis that is running at the same time, possibly for a long time. Nonetheless, we obtained crucial speed-ups this way on our analysis of the OPC UA protocol [*DHK25a*] with ProVerif.

Modular Model and Adaptive Proof Heuristics and Helpers First, let us stress that to produce such a large scale of proof obligations, it is out of the question to manually write the associated models one by one. Instead, we develop a single model with pre-processing allowing us to simply configure each of the instances corresponding to the proof obligations. In practice, we developed macro-processing Jinja or M4 and scripts (Python or bash) to produce the models, and Python scripts to call the prover, and process the results. Finally, some verification goals might not run sufficiently fast, or may not apparently terminate, or may return false attacks (in case ProVerif is used as prover). This is a well-known problem in our community and various techniques exist to mitigate such problems. Interestingly, since our methodology relies on modular models —prerequisite to allow modeling all of these threat models and protocol variants— we can easily include goal-specific proof helpers. We stress that these helpers are in the user-space of the provers and were designed to always preserve soundness of the results obtained with them.

Pattern	Message	Destination Level	STM (secrecy for the sender)
IXX	2	3	$(R_{\mathrm{re}} \wedge (R_{\mathrm{rs}} \vee R_{\mathrm{e}})) \vee (R_{\mathrm{e}} \wedge R_{\mathrm{s}}) \vee (\mathsf{active} \wedge (D_{\mathrm{re}} \wedge (R^{<}_{\mathrm{rs}} \vee R^{<}_{\mathrm{e}} \vee D_{\mathrm{rs}})) \vee (D_{\mathrm{rs}} \wedge R^{<}_{\mathrm{re}}))$
X1X	3	5	$(\mathsf{active} \wedge ((D_{\mathrm{rs}} \wedge R^{<}_{\mathrm{re}}) \vee (D_{\mathrm{re}} \vee (R^{<}_{\mathrm{rs}} \vee D_{\mathrm{rs}}))) \vee R_{\mathrm{e}} \vee (R_{\mathrm{re}} \wedge R_{\mathrm{rs}})$

Table 5.1.: Strongest threat models (STM) for secrecy of some messages from the point of view of the sender, to illustrate Example 5.5.1.

5.5. Example of Results

The methodology presented in earlier sections allows us to efficiently "evaluate" a full lattice of verification goals. As explained in Section 5.2, it is possible from this to easily extract (i) the maximal nodes that were proven false that compactly express all the minimal ways to violate the security properties of the protocol and (ii) the minimal nodes that were proven true that express the minimum set of assumptions about the adversary and the protocol for each of the security property to be guaranteed.

We gave an example for (ii) for a Noise property in Example 5.2.1. Interestingly, our Noise analysis revealed that for all secrecy and agreement properties and for all the 53 patterns we analyzed, only 74 STMs were produced. Table 5.1 shows two examples of such STMs, which we discuss further in Example 5.5.1. This enabled us to establish various results on the Noise protocols [*Gir+20*], some of which we briefly mention next.

Refining the Noise levels The Noise protocol specification lists informal payload security properties called *levels* for each payload message of a handshake (*e.g.,* Examples 2.2.2 and 2.2.3). Correctly mapping a formal security model to informal prose is generally challenging. Our systematic lattice-based analysis enabled us to automatically derive a formal definition of the Source and Destination Levels and find some flaws and counter-intuitive properties. First, in contrast with the informal Noise levels [Per18], our results have precise formal definitions, are machine-checked and considerably more granular. Further, our approach *objectively and automatically* assigns a formal meaning to the original levels as a special case. Our results also uncovered several shortcomings of the Noise levels [Per18]. First, even though the levels appear to get stronger monotonically, as suggested by the Noise specification, we find that this is not actually the case (see Example 5.5.1). This is surprising and can lead to misguided protocol choices in practice. Second, we explain why the levels, as specified in [Per18], implicitly assumed that ephemeral keys cannot be compromised, which considerably weaken these guarantees. Finally, in contrast to the 9 Noise levels, we provide 74 distinct levels (74 STMs) and show in [*Gir+20*] why this increased precision is crucial to well-informed protocol choices.

Example 5.5.1 (Non-monotonic secrecy upgrade) Consider the choice of a protocol with the goal of transmitting one payload from Alice to Bob. For the designer, only the secrecy of this payload from Alice's point of view matters. In the initial setup, both parties have a static key (but do not know their peer's static key in advance) and no Pre-Shared Key (PSK) is available.

We consider a scenario where Alice's ephemeral key can be revealed (i.e. Alice's device has a weak or faulty RNG) and the attacker is passive. Obviously no security is possible in the scenario where one party is fully compromised so we exclude it and we get the following threat model: $t = (R_{\mathrm{e}} \wedge R_{\mathrm{s}}) \vee R_{\mathrm{re}} \vee \mathsf{active}$.

With Alice as the recipient, we see in Table 5.1 that the second message of pattern IX (labeled Destination Level 3) fulfills our requirements. Upgrading our guarantee from Level 3 to Level 5 should be an improvement, so we alternatively consider sending the sensitive payload as message 3 of pattern X1X with Alice as the initiator, which is labelled Level 5. However, under the above threat model t, this payload is not secret with X1X, since there is an attack when Alice's ephemeral key can be revealed (R_e). This means that an *upgrade* from a Level 3 to a Level 5 leads to an attack on secrecy, which the Level 3 handshake message would have prevented. ◊

Selecting the Best and Identifying Redundant Protocols Using our results, we automatically compared almost all Noise protocols listed in the specification and produced a hierarchy thereof based on how the STMs compare for each property. Such a hierarchy is shown in Figure 2.11, see [*Gir+20*] for more. Using this hierarchy, we provide guidelines on which Noise protocol to choose, given a setup that describes what PKI or symmetric keys are available, and the expected range of adversary capabilities (threat model). We also identify redundant handshakes, which provide fewer guarantees than other handshakes, given the same setup. With regards to the Noise specification, we properly separate the threat-model assumptions, security goals, and monotonicity of security properties between handshakes. This allows practitioners to evaluate their environment assumptions independently of the goals they want to achieve, and enables them to pick the appropriate protocol required for their use-case.

Further Results and Recommendations for Noise We make further contributions to the Noise framework and its application. An example thereof comes from our analysis of Noise protocols using a PSK. If a (publicly known) *dummy key* is used as PSK (a suggestion made in the specification), we show that, surprisingly, some protocols provide incomparable levels of security when using a dummy PSK compared with when using no PSK at all. Another example concerns anonymity, where our results reveal a missing requirement related to the handling of session identifiers.

Other Analyses Figure 3.2 already presented some of our hash weakness analysis results [*Che+23*]. The lattice approach was critical to tame the scale of the proof effort otherwise needed. Similarly, we could not have achieved our OPC UA analysis [*DHK25a*] without leveraging this approach, where some single proof obligations requires 10 hours for ProVerif to conclude. Interestingly, we took inspiration from [BC14] to propose a first version [*Gir+20*] that we later refined in [*Che+23*; *DHK25a*]. It has since been used by others [Laf+25; LMR24; Jac+23; Mah25].

5.6. Proof Helper Methodology for ProVerif [*DHK25a*]

We mentioned proof heuristics and helpers we can easily leverage in a proof obligation-specific manner when using the lattice approach (see also Section 2.4.2). We did so with standard techniques for our Noise [*Gir+20*] and Hash weaknesses [*Che+23*] analyzes; Tamarin oracles and helping lemmas [Bas+19] as well as noselect, nonce contexts, and helping lemmas for ProVerif [BSC16].[6] We now present a more advanced methodol-

6: See the paragraphs dedicated to proof guidance in Sections 2.4.1.3 and 2.4.2 for more details on these techniques.

ogy we developed for our OPC UA analysis [*DHK25a*] to thwart proof search loops in ProVerif, which is a recurring issue in complex protocol verification.

OPC UA has a complex state-machine where clients and servers first create a *channel* protecting further communication, they can then create *sessions* in such channels, which can then be *activated* by users on clients. (We defer the details of these mechanisms in Chapter 6 and keep an informal presentation here.) In practice, despite the addition of standard proof helpers during our OPC UA analysis, the verification with ProVerif did not terminate for authentication properties on most protocol configurations. Introspection of the ProVerif's proof search procedure revealed that it was looping. The loop resulted from the handling of session information stored in tables —necessary given the complex state machine— and correspond to actual loops in the protocol state machine. For instance, in order for a user to (re)activate an existing session, a table entry of a previous session is necessary and a new entry of the updated session is created. Hence, insertion of a table entry requires the existence of a table entry. We therefore applied the following principled methodology that consists in two steps and that is quite effective at thwarting such loops, which is a recurring issue in complex protocol verification. We believe this is of general interest and could be reused for other ProVerif analyses. A more detailed presentation with a running example is given in [*DHK25a*, §D.2].

Breaking loops. ProVerif's Horn clauses resolution procedure is guided by a *selection function* which can be modified using a `noselect` statement. Manually inspecting the detailed output, *i.e.,* created clauses displayed with the option `verboseRules`, we are able to identify the facts that are almost always selected and that cause the loop, and *break* this loop by adding `noselect` statements on those facts. In our case, as explained above, the loop was caused by table accesses. We therefore added statements of the type `noselect table(S_sessions([...]))` for session (and channel) tables to deprioritize the selection of a given table fact and avoid entering the loop.

Adding invariants. However, as tables are not selected anymore, this systematically results in cannot-be-proved. Indeed, without the ability to trace back from where a table entry originates, ProVerif conservatively assumes tables could initially contain arbitrary entries, leading to "false attacks". Fortunately, ProVerif explicitly describes those "false assumptions" it made in the *goal reachable*, *i.e.,* the clause that falsifies the property. One can then contradict these with invariants, confirm they are correct by proving them with ProVerif, and then retrying to prove the initial property assuming the invariants. In short, such invariants "compensate" for the unexplored part (tracing back the origin of a table entry). This is an iterative and generic process, where invariants are added when needed by inspecting the false assumptions that ProVerif generates.

For instance for Open Platform Communication Unified Architecture (OPC UA), when a client receives a message in a session, associated to some channel, it is signed with the server's signing key. This signing key is the only known link between the session entries on the client and server side that ProVerif infers (as the `noselect` avoids resolution on the `table` facts). In this case we can provide an invariant which states that if the client and server session table entries use channel keys that coincide then these session entries also coincide on the client and server identities and all the channel information (unless the client long-term key was leaked).

These invariants are first proved without the noselect statement and then declared as axioms, *i.e.,* lemmas that are admitted, when proving the security property. Finally, we manually check that the dependency graph between invariants is not cyclic (*i.e.,* we never assume inv1 to prove inv2 and assume inv2 to prove inv1). In our case, we provide this graph in [*DHK25b*]. We stress that, for a single iteration, where some invariants are assumed to prove other invariants, proof helpers can additionally be used and can be specific to this iteration. We extensively used this opportunity to, for instance, fine-tune the context of nonces, whose required precision greatly varies with the invariant or property under study.

We heavily relied on this technique to thwart proof search loops caused by session re-activations, channel switchings, *etc.* The proof of the authentication properties required the addition of 10 noselect statements and of more than 100 lemmas.

Part B.

Large-Scale Protocol Analyses

Contents of this Part

6 Repairing an International ICS Standard 75

6.1 Open Platform Communication Unified Architecture (OPC UA) Protocol 77

6.2 OPC UA Formal Model 83

6.3 Analysis Results 84

6.4 Lessons Learned and Discussions with the OPC Foundation . 94

6.5 Conclusion 97

7 Challenging and Improving the 4G and 5G Mobile Communication Standards 99

7.1 Simplified Overview of 5G Authentication and Key Agreement (AKA)100

7.2 Formal Analysis of 5G AKA 104

7.3 New Privacy Risks and Practical Attacks 109

7.4 Lessons Learned and Discussions with GSM Association (GSMA) and 3rd Generation Partnership Project (3GPP) 115

8 Breaking and Fixing E-voting and French National Election122

8.1 Introduction 122

8.2 Context 124

8.3 Reverse the Protocol 127

8.4 Vulnerabilities, Attacks, and Fixes 130

8.5 Recommendations and Lessons Learned 139

8.6 Breaking and Fixing Other E-Voting Components 141

8.7 Responsible Disclosure and Further Impact 142

8.8 Conclusion 145

Table 5.2.: Summary of the impact of the attacks we found on the e-voting protocols used for politically-binding elections in France in 2023 (French Legislative E-Voting Protocol (FLEP)). We found out that neither of the central security goals in e-voting —verifiability or ballot privacy— could be guaranteed under classical trust assumptions. More precisely, this Table shows a comparison between the threat models under which we found the FLEP is breached versus the strictly stronger threat model under which state-of-the-art protocols such as Belenios are secure [CGG19]. Legend: ✓ = can be untrustworthy, ~ = must be assumed trustworthy (as expected), ✗ = must be assumed trustworthy due to our novel attacks (unexpected). More details about the threat model and the attacks are given in Chapter 8.

Security Goal	Voter	Voting Device	Plaintext Channel	Voting Server	Decryption Authorities	3rd-party
Our attacks on the FLEP falsify the properties under the threat model:						
Verifiability	~	~	✗	✗	~	~
Ballot privacy	~	~	✗	✗	~	~
State-of-the-art protocols such as Belenios are secure under:						
Verifiability	~	~	✓	✓	✓	~
Ballot privacy	~	~	✓	✓	~	~

This Part presents three of our main large-scale protocol analysis results and their impacts. In Chapter 6, we present our analysis of an international Industrial Control Systems (ICS) standard that led to the discovery of multiple vulnerabilities, their assignment as CVEs, and their subsequent fixes now included in the standard. In Chapter 7, we summarize our work on identifying new risks and practical attacks in the international mobile communication standards for 4G and 5G, as well as the responses of the industry to these attacks, including the modification of the 5G standard to prevent a new privacy attack we identified. Finally, in Chapter 8, we present our work on breaking and fixing e-voting protocols, in particular the one used in French national elections in 2023, for which we responsibly disclosed multiple vulnerabilities to the relevant French authorities, leading to the modification of the protocol with our fixes for the subsequent elections.

Each of these chapters presents large-scale protocol analyses for which the formal description of the protocols and of the security goals is a contribution in itself. Indeed, these systems are either opaque and lack specification, calling for reverse-engineering efforts (see Chapter 8), or are at best described in natural language in lengthy documents[7] (Chapters 6 and 7). In both cases, formalizing them is a significant effort that is a prerequisite to any formal analysis. In particular, the specification for such systems rarely includes precise threat models or security goals, which we had to derive ourselves from informal descriptions. Moreover, the protocols are often specified through multiple documents describing different functionalities or components in isolation at a rather low level of abstraction. It takes a significant effort to piece these documents together and obtain a coherent protocol-level description of the entire system with the different flows of messages, internal state-machines, verification carried out on incoming messages, *etc.*

Next, for most of the analyses discussed in this Part, we used formal verification tools to obtain strong guarantees on the security of these systems or to automatically discover attacks. In all these cases, state-of-the-art tools could not be used out-of-the-box at the time to obtain meaningful results. We had to extend these tools and verification algorithms, develop new proof methodologies and verification strategies. We will not discuss these aspects here, some of them are covered in Part A. Instead, we focus on the results of these analyses and their significance.

7: Our analysis of the 3GPP 5G standard [*Bas+18*] required us to go through 722 pages across four 3GPP Technical Specification documents. For our formal analysis of OPC UA [*DHK25a*], we have read five specification documents (out of more than twenty) totaling 528 pages.

These analyses led to the discovery of multiple vulnerabilities, which we responsibly disclosed[8] to the affected stakeholders, and to the proposal of effective fixes that we also formally verified when possible. As we shall see, these results had a significant impact on the security of widely used systems. For instance, the central authentication protocol (AKA) running in all 4G and 5G mobile phones worldwide has been modified to prevent a new privacy attack we identified. The specification of the international ICS standard we analyzed (OPC UA) has been revised to fix multiple vulnerabilities we found. Finally, the e-voting protocol used in French national elections has been modified to prevent critical attacks we identified on the version used in the 2023 elections.

Additionally, following each of these analyses, we stepped back and drew lessons and future directions for practitioners, designers, and researchers. We also document the responsible disclosure process we followed for each of these analyses and reflect on the challenges we faced in this process. We would also like to emphasize the importance, for the research community, of communicating such novel attacks when accompanied by lessons and insights about these systems and their designs.[9] This is in line with David Basin and Srdjan Čapkun's position paper about "The Research Value of Publishing Attacks" [BC12] that we fully share:

> As our physical and digital worlds become more tightly coupled, the incidence of attacks will increase as well as their consequences. Many of these attacks will be newsworthy, but most will not be research-worthy. This does not mean papers featuring attacks on highly visible systems should not find their way into research conferences; having had such papers published, the authors of this column do appreciate that the community accepts results of this kind. However, as researchers we should have high aspirations. With every attack paper there is an opportunity to truly contribute to the community with new insights into both systems and their vulnerabilities, and adversaries and their capabilities. We believe that one should take this opportunity and, after discovering an attack, take a step back and reflect on what can be learned from it, and afterward present it to the community.

8: We systematically addressed ethical concerns throughout this line of work. When finding vulnerabilities, we followed responsible disclosure processes with embargo periods of two to three months, coordinating with vendors until fixes were deployed before publication. When analyzing deployed systems, we limited ourselves to passive observations to avoid disrupting services or affecting users. Testing experiments targeted only local test instances, never production deployments.

9: We are also aware of the risks of purely chasing CVEs and attacks in a sensationalist, time-sensitive, technology-focused way, and have tried to move away from this approach, which we believe is detrimental to lasting scientific contributions and to the community (see Section 11.4.0).

Repairing an International ICS Standard

6.

🔍 *Focus on a Comprehensive Formal Security Analysis of OPC UA [DHK25a]*

ICS manage critical infrastructure facilities, such as power plants, and are part of Operational Technology (OT), *i.e.,* covering devices that interact with the physical environment [Sto+23]. As opposed to classical Information Technology (IT) systems, OT historically prioritizes safety and reliability, above other security properties such as confidentiality. Moreover, the life cycle of OT systems is much longer, as updates require heavy and costly qualification processes. Severe security incidents, *e.g.,* the Stuxnet [BR17] worm in 2010, led to adding a security layer (such as Internet Protocol Security (IPsec), or more often Transport Layer Security (TLS) [OSS16]) to existing ICS protocols. However, these additional security layers are tailored for the Internet and are poorly suited to OT, motivating secure ICS protocols.

OPC UA is a general purpose ICS protocol between Supervisory Control and Data Acquisition (SCADA) systems – including operators' workstations – and automation devices, that regulate an industrial process through sensors and actuators. It is developed by the OPC Foundation and standardized as IEC 62541 (currently in v1.04) [*DHK25a*]. OPC UA has undergone several security analyses by state agencies, such as the German BSI (two analyses in 2017 and 2022 [Dam+17; Dor+22]), the French ANSSI (development of the open source implementation S2OPC[Sys23]), as well as by OT security companies that analyzed implementations [CT18; Tea23; BM23].

OPC UA follows a modular and layered design, where functionalities are activated through profiles. We focus on the OT profile "UA-TCP UA-SC UA-Binary", which is by far the most commonly used [Dor+22, § 6.3.4]. This so called OPC UA Binary profile mainly relies on the UA Secure Conversation (UASC) sub-protocol. The protocol is parameterized by a *security policy*: policies based on Elliptic Curve Cryptography (ECC) have been introduced in version 1.05 of the OPC UA specification. The use of ECC based Diffie-Hellman is supposed to provide stronger confidentiality guarantees, namely Perfect Forward Secrecy (PFS), compared to RSA which was the only option previously. In addition, ECC allows for improved performance through shorter keys, which can be crucial for embedded systems. Once a secure channel has been opened, sessions can be created inside this channel. Users can then activate sessions using their credentials and send user requests. OPC UA also allows re-opening existing channels (for key rotation), and re-activating sessions (for transferring to another channel or user).

The OPC UA Foundation designed the OPC UA protocol to be both secure-by-design and compatible with the unique features of OT, and thus went for a complete re-design from scratch instead of using well-established cryptographic protocols. Hence, the following research question arises:

> *What are the actual security guarantees provided by the OPC UA protocol? Is it possible to formally establish these guarantees, and*

6.1 OPC UA Protocol 77
6.1.1 Overview 77
6.1.2 Protocol Description . . . 78
6.1.3 Security Goals 81
6.2 OPC UA Formal Model 83
6.2.1 Modeling in the Applied Pi Calculus 83
6.2.2 Formal Security Properties 84
6.3 Analysis Results 84
6.3.1 Client Impersonation in ECC 85
6.3.2 KCI: User Impersonation . 88
6.3.3 Session Hijack by Reopening or Switching 90
6.3.4 KCI: Session and User Confusion 91
6.3.5 Other Findings 92
6.4 Lessons Learned and Discussions with the OPC Foundation 94
6.4.1 Key Lessons on Protocol Design 94
6.4.2 Fixes of the Main Remaining Attacks in v1.05.07 . . 94
6.4.3 Responsible Disclosure and Actions Taken by the OPC Foundation 95
6.5 Conclusion 97

under which assumptions?

Contributions In [*DHK25a*], we perform the first comprehensive, formal security analysis of OPC UA. We leveraged the ProVerif protocol prover [Bla01] to conduct this automated analysis. Our main contributions are as follows.

[*DHK25a*]: Diemunsch, Hirschi, and Kremer, *A Comprehensive Formal Security Analysis of OPC UA*, USENIX'25

We provide a *detailed description* of the security sub-protocols of the OPC UA Binary specification version 1.05 (Sections 6.1.1 and 6.1.2), their expected security goals, and the considered threat model (Section 6.1.3). The relevant security aspects of the specification [*DHK25a*] span 5 documents (out of more than 20), totaling 528 pages. We have extracted what we believe to be a specification suitable for a protocol security analysis. This description is concise, yet comprehensive as it covers both ECC and RSA security profiles (also covering v1.04), sub-protocols for both channel and session establishment, and management, under different configurations (considering different channel modes, session security modes, and user authentication means).

In Section 6.2, we present our *formal model of OPC UA* in the applied π-calculus which in particular encompasses the following features: *(i)* the reopening of secure channels, *(ii)* the reactivation of sessions using passwords as well as user certificates, *(iii)* the switching of sessions between secure channels, *(iv)* the authentication of user, client and server involved in a request and its response. This involves dealing with a complex state machine.

We *formally specify the security properties* (authentication and confidentiality) in Section 6.2.2. The properties are often tricky to formalize, and we need to condition them by the exact configurations in which a property is expected to hold, as the protocol also allows degraded configurations (that may be deployed simultaneously in other sessions). This also means that our model precisely describes in which configurations and threat models each property is actually expected to hold.

We *analyze OPC UA* using ProVerif. Given the size and the complexity of our model, and the large number of possible configurations, we provide tooling[1] that allows us to *(i)* generate ProVerif models for a particular set of configurations, and threat model; *(ii)* efficiently explore the lattice of configurations and threat models, and automatically find maximal configurations in which a property holds, as well as minimal configurations in which we find attacks (or reach the limits of ProVerif). Moreover, we used many advanced features of ProVerif [BCC22] to fine tune the model and guide the proof search, to be able to conclude in complex configurations. Some of these proof techniques and methodologies are described in Chapter 5, see [*DHK25a*] for more details. We believe that this is among the most complex analyses performed with ProVerif, both due to the size of the model as well as the complexity of the state machine, *e.g.,* the model requires a lot of information to be stored in a global state. To the best of our knowledge, it is the largest ProVerif model in terms of LoC and initial clauses (the internal protocol representation on which ProVerif reasons) ever analyzed. All models, results, and instructions to reproduce them are provided in our artifact [*DHK25b*].

1: Our proof methodology and tooling are briefly presented in Sections 5.1.3 and 5.6. See [*DHK25a*, §4] for more details.

Our analysis allowed us to discover 8 *new vulnerabilities and other weaknesses* in OPC UA v1.05 (Section 6.3), 6 of these also affect v1.04. Each of these has been responsibly disclosed to and acknowledged by the OPC UA Foundation, which assigned Global CVE Allocation System (GCVE) identifiers to them.[2] We proposed provably secure fixes and other

2: GCVE is a decentralized vulnerability identification system launched in 2025 by CIRCL (Luxembourg) as an alternative to the MITRE-operated Common Vulnerabilities and Exposures (CVE) program. Each GCVE Numbering Authority, such as the OPC Foundation, allocates identifiers in its own namespace, without going through a central authority.

mitigations, most of them are now included in the specification. Finally, we draw more general lessons for the future of OPC UA and discuss our responsible disclosure process with the OPC UA Foundation and reflect on the actions they took in response to our work in Section 6.4.

6.1. OPC UA Protocol

6.1.1. Overview

We start by presenting the OPC UA protocol version 1.05.03 with a high-level overview of the sub-protocols and their interactions. In this work, we cover the security sub-protocols, namely the secure channel and the session sub-protocols. They are specified in the OPC UA standard, mostly in [OPC23, Parts 4, 6, 7]. They involve three kinds of agents: *clients* (*e.g.,* user workstations), *servers* (*e.g.,* SCADA), and *users* (*e.g.,* humans operating user workstations). Each agent is assumed to be enrolled in a Public Key Infrastructure (PKI) and possesses a certificate that describes its identity and role (*e.g.,* a client certificate C_{cert}) with the associated private key (*e.g.,* C_{sk}); users store them on a smart card or may alternatively use a login password pair.

The main flow is depicted in Figure 6.1: a client *opens* a *secure communication channel* with a server and *creates* a *session* inside this channel. A user on this client can then log in, *i.e., activate* a previously created session, and use this session to send/receive *requests* to/from the server. We now briefly present those sub-protocols.

Secure communication channels. Any communication between a client and a server in OPC UA starts by opening a secure channel. While there exists a specific configuration (OPC UA HTTPS) that leverages TLS for this, the more widely deployed profile (OPC UA Binary) has a dedicated channel sub-protocol called UASC. This sub-protocol is made of two parts: *(i)* a handshake layer, called *OpenChannel* (see Figure 6.1), that establishes a new channel ID id_c and a set of symmetric keys sk, shared between client and server; *(ii)* a record layer that protects all subsequent messages with these symmetric keys (depicted by $\|.\|^{id_c,sk}$). The symmetric keys can be renewed by re-running OpenChannel on an existing channel.

Channels can be configured in three Modes:

- Enc: provides integrity and confidentiality.
- Sign: provides integrity only. This can be required for the system to be monitorable by third parties.
- None: insecure channels, which are reserved for scanning the network and debugging.

Finally, a channel SecurityPolicy defines the asymmetric and symmetric cryptographic algorithms to use. The protocol supports two families: RSA and ECC. The ECC family offers an Elliptic Curve Diffie–Hellman (ECDH) key exchange that can additionally provide PFS.

Session creation and activation. OPC UA sessions provide a user context for requests and their responses.* Sessions are created by a client in a channel (see CreateSession in Figure 6.1), and yield a session identifier id_s and a session token tok_s (explained in Section 6.1.2.2). The options (SSec | SNoAA) define the security level provided by the session and will

* There are also sessionless requests, but we choose not to cover them in this work because they rely on issued tokens (see next footnote).

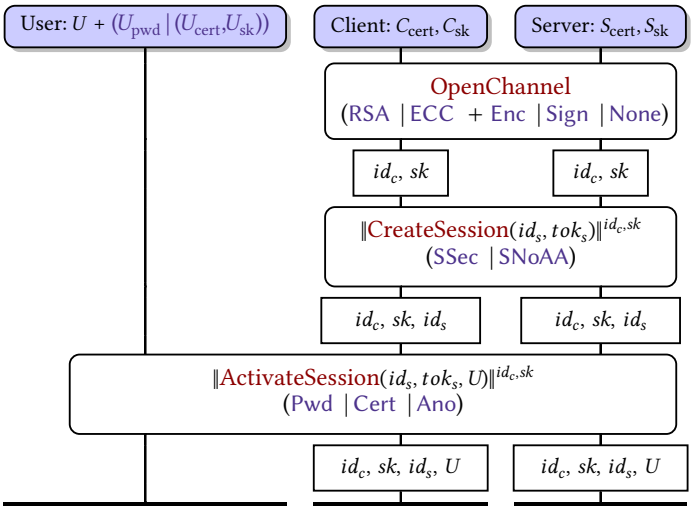


Figure 6.1.: Overview of the OPC UA protocol: a client opening a secure channel (id_c with keys sk), creating a session id_s, and a user U activating it. Sub-protocols, which will be detailed later, are indicated in red, and protocol configurations in blue. $\|\cdot\|^{id_c,sk}$ denotes channel protection, that depends on the channel configuration (Encryption&Signature | Signature | None).

be explained later in Section 6.1.2.2. Created sessions can be activated (see ActivateSession) directly by the client with the anonymous user (Ano), or by a regular user. Regular users rely for this on login password pair (Pwd) or a certificate (Cert), which may be stored on a smart card for better security (see NIST[Sto+23] § 6.2.1.4.3).†

State machine. There exist various ways for channels and sessions to evolve and the resulting state machine, depicted in Figure 6.2, is rather complex and further complicated by the various configurations that can be arbitrarily combined: (RSA | ECC + Enc | Sign | None + SSec | SNoAA + Pwd | Cert | Ano). Once opened, a channel can be *Reopened*, by running OpenChannel on the existing channel, resulting in a key renewal. Sessions can then be created and activated on any channel. By running ActivateSession on a previously activated session, users can:

- *ReActivate*: hand over sessions to other users (*e.g.*, at shift change),
- *Switch*: transfer them to a new secure channel (*e.g.*, in case the previous channel was terminated).

Such complex operations are required for increased reliability, which is key in OT systems (see [*DHK25c*]).

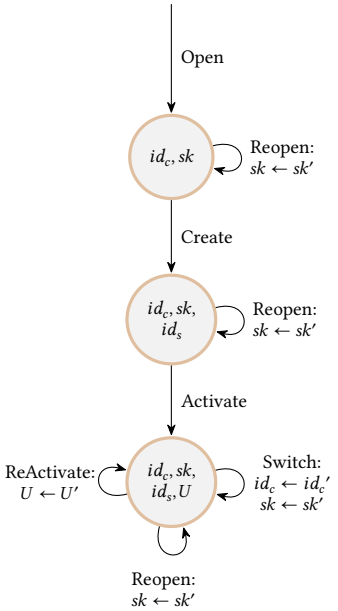


Figure 6.2.: Overview of the OPC UA state machine

6.1.2. Protocol Description

For the sake of clarity, we first present the different sub-protocols when SecurityPolicy is set to ECC and Mode to Enc, and then explain the main differences with other configurations. We refer the reader to [*DHK25c*] for their detailed description.

Cryptographic Schemes. We use the following notations for cryptographic operations. $[\![m]\!]_{sk}$ denotes the signature of m with the private key sk, $\{m\}_k$ the symmetric encryption of m with the key k, and $[m]_{mk}$ the addition of a Keyed-Hash Message Authentication Code (HMAC) with key mk. The choice of the precise cryptographic suite implementing those depend on server configuration. *E.g.*, for SecurityPolicy [ECC-B] ECC-nistP256 [OPC24], they are respectively instantiated with ECDSA-SHA2-

† Another user authentication mode uses issued tokens (*e.g.*, WS-SecurityTokens (Kerberos), JSON Web Token (JWT) or OAuth 2 tokens). They rely on authorization services, that are protocols in their own rights. However, issued tokens details are neither specified nor included in the security profiles of version 1.05. Hence, we do not cover them here, and rather focus on the two core OPC UA user authentication methods.

256, AES128-CBC and HMAC-SHA2-256. More importantly for this work, the combination of HMAC and symmetric encryption roughly follows the MAC-then-Encrypt construct with some data in plaintext. Formally:

$$[m, \{p\}_{ek}]_{mk} := m, \mathsf{AES}_{ek}(p, \mathsf{HMAC}_{mk}(m, p)).$$

The case of profiles with authenticated encryption schemes (a minority of profiles: 3 over 11) is a bit different (see [*DHK25c*]) and is left as future work.

6.1.2.1. UA Secure Conversation

We now present the UASC sub-protocol, including OpenChannel, see Figure 6.3. The first two messages are for channel opening or reopening, and the last two are for requests and responses sent over the created channel. The requests that can be sent over the channels are described in Section 6.1.2.2.

Channel (re)opening. To open a secure channel, the client sends an *OpenSecureChannel* request. This request notably includes the client's certificate C_{cert} and is signed with the associated private key C_{sk}. The channel ID id_c^0 is initially set to 0 to indicate that a new channel should be opened. There is also a counter Rq, incremented at each request. Finally, the request includes the Diffie-Hellman (DH) half-key $C_{\text{hk}} = g^c$, from which the shared secret will be derived.

The server checks the client certificate and the signature of the client request. Its response is similar, but includes its own certificate S_{cert} and half-key $S_{\text{hk}} = g^s$. The server also chooses a new, fresh channel ID id_c (a counter) and tok_c (a nonce). The former identifies the channel, while the latter identifies the shared secret sk that will be derived. sk is computed by a Key Derivation Function (KDF) of the shared DH secret $g^{c \cdot s}$. More precisely, two keys for encryption ($sk.C_{\text{esk}}$ and $sk.S_{\text{esk}}$) and HMAC ($sk.C_{\text{msk}}$ and $sk.S_{\text{msk}}$) are derived. The client checks the server certificate and the signature of the response and derives sk as well. (id_c is included twice.)

An existing channel with ID id_c^{old} can be reopened by a client by choosing $id_c^0 := id_c^{\text{old}}$ (instead of zero). The server will then choose a new security token tok_c' but keep $id_c = id_c^{\text{old}}$. The client must reopen before the channel lifetime (depending on server configuration) expires.

Message security. Once a channel is open, client and server can send requests and responses, which are wrapped and cryptographically protected as shown in the last two messages of Figure 6.3. In particular, reusing notation from Figure 6.1,

$$\|\text{request}\|^{id_c, sk} := [\text{MSG}, id_c, tok_c, \{Rq', \text{request}\}_{sk.C_{\text{esk}}}]_{sk.C_{\text{msk}}}.$$

Other configurations. When SecurityPolicy = RSA, the client and server exchange nonces C_{nonce}^c and S_{nonce}^c instead of DH half-keys. Those nonces are asymmetrically encrypted with the receiver's public key and are sent along with the hash of the public key used for encryption. When Mode = Sign, the encrypted parts are in plaintext in MSG messages. When Mode = None, signatures, HMAC, encryption, certificates, and half-keys (or nonces) are omitted in all messages. See a detailed description of those configurations in [*DHK25c*].

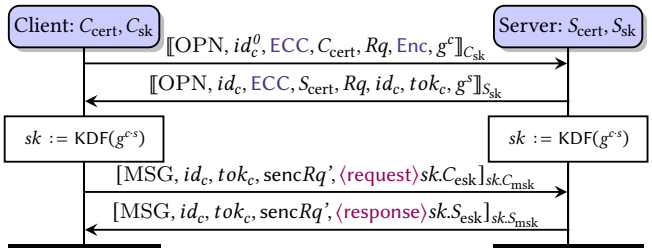


Figure 6.3.: (Re)Open Secure Channel (first two messages) and Channel Communications (last two) in (ECC + Enc). OPN is a header string. After the first two messages, client and server can derive a set of symmetric keys sk containing encryption keys C_{esk}, S_{esk} and HMAC keys C_{msk}, S_{msk}. The last two messages describe the channel protection of subsequent requests/responses payloads ⟨request⟩/⟨response⟩ (cf. Section 6.1.2.2).

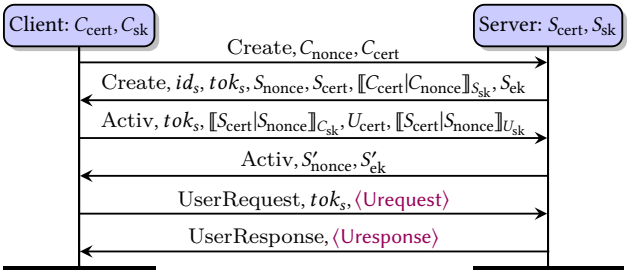


Figure 6.4.: Session creation (first two messages) and activation (third and fourth) and user requests/responses (last two) in (ECC + Enc + SSec + Cert). The session activation involves the user to produce the signature with U_{sk}. Create and Activ are header strings. The last two messages describe how user requests ⟨Urequest⟩ and responses ⟨Uresponse⟩ payloads, from the application layer, are wrapped in a session.

6.1.2.2. Sessions

We now describe the CreateSession and ActivateSession sub-protocols (both depicted in Figure 6.4). The first two messages allow the creation of a session, the third and fourth its activation, and the last two describe how *user requests*, at the application layer, are wrapped in a session. We stress that all those messages are sent over and protected by some channel, and act as ⟨request⟩ and ⟨response⟩ of that channel, cf. Figure 6.3.

Session creation. The client sends a Create request with a fresh nonce C_{nonce} and C_{cert}. The server checks that the client certificate is the same as the one associated to the channel through which this message is sent (this client certificate was sent in the OpenChannel request). The server response provides a new, fresh session ID id_s (a counter) that will identify the session. It also includes a fresh *Session Authentication Token* tok_s (a nonce) that serves the same purpose as id_s but is supposed to be kept secret. A fresh server nonce S_{nonce} is provided and acts as a challenge to the client for the next session activation. The server also proves possession of their certificate, by signing C_{cert} and C_{nonce} with S_{sk}. Finally, a signed fresh DH half-key $S_{ek} = [\![g^s]\!]_{S_{sk}}$ is added in an additional field for later use, should the session be activated in configuration Pwd (described next). If all checks pass, clients and servers notably store the session information: id_s, tok_s, id_c (id_c refers to the channel on which the messages were sent).

First activation. Once created, a user can *activate* a session (id_s, tok_s, id_c) by sending an Activ request. This request includes tok_s and a signature of

S_{cert} and S_{nonce}, sent in the previous session creation response, to prove possession of C_{sk}. In mode (Cert) (as in Figure 6.4), the request must also contain the user's signature (with U_{sk} associated to U_{cert}) of S_{cert} and S_{nonce}. In practice, the signature can be computed on the user's Smart Card. The server verifies the signatures and responds with fresh S'_{nonce} and S'_{ek} for future reactivations.

User requests and responses. User requests ⟨Urequest⟩, produced by the application layer, are wrapped in a session by appending the session token tok_s. The user response ⟨Uresponse⟩ can then be linked to that session as well, since the response will use the same Rq at channel level. Indeed, recall that such session requests are additionally sent through, and cryptographically protected by, some secure channel.

Reactivation and channel switching. Session handover to another user is achieved by sending an activation request referring to a previously activated session, on the same client and channel, but for a different user. Similarly, the switching of secure channel is an activation of the same session, on the same client by the same user, but carried out on a different secure channel, with the same security Mode and the same client and server certificates.

Other activations. In configuration Ano, U_{cert} and the signature with U_{sk} are omitted and the session is not bound to a user. In configuration Pwd, U_{cert} is replaced by a login and the user's signature is replaced by the password encrypted with a symmetric key derived from the latest S_{ek} and some fresh client DH half-key, both sent together with this ciphertext. When Mode = RSA, the only difference is that S_{ek} is omitted and, when necessary, the password is instead asymmetrically encrypted with the server's public key. Note that in configuration Enc , the password *may* be sent without symmetric encryption (*i.e.,* sent as plaintext at the session layer), as encryption is provided at the channel level.

Session security. We now explain the differences between SSec and SNoAA. By default, in SSec, server and client certificates are exchanged and trust checked against the PKI, at session creation and activation. However, the specification also defines a *No Application Authentication* mode [OPC23, Part 7, § 5.3], where the server is configured to accept all certificates and only use them for message security. We call this mode SNoAA. We provide further details about SNoAA and the other aforementioned configurations in [*DHK25c*].

6.1.3. Security Goals

We now describe the security goals of OPC UA, and the threat model under which these goals should hold as described in [OPC23, Part 2 *"Security Model"*]. In [*DHK25c*], we provide more evidence, with references to the specification, supporting our interpretations.

Threat Model We first summarize the threat model in terms of attacker capabilities and compromise scenarii detailed in [OPC23, Part 2, § 4.3 *"Security threats to OPC UA systems"*]. OPC UA is claimed to resist the following threats: eavesdropping (§ 4.3.3), message spoofing (§ 4.3.4), message alteration (§ 4.3.5) and replay (§ 4.3.6), emitting malformed messages (§ 4.3.7). The attacker is also able to set up a rogue server (§ 4.3.10), and to act as a Machine-in-the-Middle (MIM) (§ 4.3.9). Therefore, we shall consider an *active adversary on the network* who can eavesdrop, intercept, manipulate, and inject messages; *i.e.,* a DY attacker.

Moreover, user credentials (passwords, or certificate private keys) may be compromised (§ 4.3.12). Client and server certificates and associated private keys are also considered as potentially compromised [OPC23, Part 4, § 6.1.4]. In short, the attacker can compromise agents' long-term secrets. Channel symmetric keys *sk* are more ambivalent: their lifetime depends on a parameter (*i.e., revisedLifetime*) spanning from a few minutes to 40+ days and is not enforced by security profiles. We conservatively consider those keys as semi-long-term. We do not consider compromise of such keys by default, but will allow such compromise in specific cases (password confidentiality), when mentioned explicitly. However, full attacker control of a machine, allowing ephemeral secret leaks (*e.g.,* DH client's secret *c*), remains out of scope [OPC23, Part 2, § 4.4].

All of these assumptions are justified in [*DHK25c*] with explicit quotes from the specification. As an illustration and example, we only explain here why we consider the presence of *compromised* agents. In particular for clients and servers, we consider the possibility that their long term private keys (associated to their certificate) have been compromised. Such leaks may happen, for instance, when a device is decommissioned without appropriate erasure of sensitive data. This is a common threat in the context of PKI, and is addressed in the specification:

> **[OPC23, Part 4, § 6.1.4 *"Creating a SecureChannel"*]** *Certificates can be compromised, which means they should no longer be trusted. Administrators can revoke a Certificate by removing it from the trust list for all applications or the CA can add the Certificate to the Certificate Revocation List (CRL)*

Security Properties We now discuss the security properties that OPC UA is supposed to guarantee as per [OPC23, Part 2, § 4.2 *"Security objectives"*] (with additional supporting evidence in [*DHK25c*]).

Confidentiality. Secure channels with Mode = Enc must provide confidentiality of data exchanged on the network: user requests and responses. Passwords must remain confidential for all channel modes. Moreover, channels configured with ECC additionally provide PFS, *i.e.,* any message exchange prior to the compromise should remain secure. To sum up, OPC UA is expected to guarantee that: *(i) The attacker cannot learn the user requests and responses payloads when sent over channels in configuration Enc. (ii) PFS is guaranteed for the payloads sent over channels in configuration (ECC + Enc). (iii) The attacker cannot learn the user passwords.*

Integrity and Authentication. Integrity and authentication of user requests are key requirements for OPC UA [OPC23, Part 2, § 4.2,5.1.1]. We express those more formally as *agreement properties* in Lowe's authentication taxonomy[Low97], where properties are expressed as relations between different agents' point of views of a protocol execution. Data and entity authentication can be expressed as a single, strong *agreement property*: *if a server S accepts a user request R supposedly from a user U on client C, then U has indeed initiated request R for server S on C*. Conversely, *if a user U receives on client C a user response R supposedly from server S, then S has indeed responded with R to client C for user U*. As the protocol is supposed to protect against replays, we moreover require that these agreements are *injective*, *i.e.*, each received request corresponds to a distinct request emission.

6.2. OPC UA Formal Model

6.2.1. Modeling in the Applied Pi Calculus

We briefly discuss our ProVerif formal model of OPC UA [*DHK25b*], and explain some of our modeling choices. As a general guideline, we strive to make conservative choices that favor the attacker. For example, when the specification allows several options we allow all of them, or choose the least secure one.

Our model is based on three main processes, one for each kind of agents (users, clients and servers), running in parallel. We consider an unbounded number of agents, and each agent may engage into an unbounded number of protocol sessions. *E.g.,* A client may start an arbitrary number of sessions with different servers. Each of these main processes contains subprocesses for specific tasks. The client process can (re)open channels and create sessions. The user process can activate sessions (using the user's credentials) and send/receive user requests/responses. The server process handles the different kinds of requests (channel (re)opening, session creation, (re)activation and channel switching, as well as user requests). As the scheduling is adversarial, the attacker decides which actions are triggered and which agents execute a given sub-protocol.

When an agent is created, it is enrolled in a PKI that we model as a trusted third party issuing certificates for clients, servers, and users. At any time, we allow the attacker to compromise an agent. Compromise of agent A triggers the event leak(A) and results in the output of A's secret key, and password if A is a user.

As illustrated in Figure 6.2, the state machine is rather complex, and yet fully captured by our model. We use tables to store the local states of the protocol agents. In particular, we use tables for recording the list of channels that were opened and sessions that were created/activated by a client or server.

Our model is designed to be modular and allows us to enable/disable the modes and features that are to be supported. We can set the SecurityPolicy (ECC or RSA), channel modes (None, Sign or Enc), the session security level (SSec or SNoAA), as well as the supported user credential modes (Pwd, Cert or Ano). For example, the configuration (RSA | ECC + Enc | Sign | None + SSec | SNoAA + Cert) supports all kind of crypto, channels, and session modes as well as user identification through certificates. Further parameters allow us to enable or disable agent and channel key compromise, reopening of channels, and channel switching. Being able to use such parameters has proven to be extremely useful to establish our proof methodology (see Sections 5.1.3 and 5.6). We stress that we aim to obtain results with respect to a rich model with almost all modes enabled (although we sometimes consider some option values in isolation).

The complexity of our model is also reflected by its size: while the ProVerif files were written in a modular way that amount to 2.4k LoC (including all declarations and property definitions), the unfolded process that ProVerif generates to reason corresponds to 8.6k LoC that are translated into 2.3k initial clauses (for the full configuration when verifying a confidentiality property). As a point of comparison the most comprehensive model of TLS 1.3 [BCW22] has an unfolded process of 7.3k LoC translating to 1.4k clauses.

6.2.2. Formal Security Properties

In this section, we explain how we formally modeled in ProVerif, the security properties identified in Section 6.1.3.

$Conf_C$ and $Conf_S$: Confidentiality of user requests and responses. When Mode = Enc, user requests and the corresponding responses are supposed to be confidential. In addition, when SecurityPolicy = ECC, the protocol is expected to provide PFS. These requirements are formalized in Figure 6.5, where $C_{data}(\cdot)$ is an event triggered by clients whenever they send/receive a user request/response R. For any user request R sent at time t in a session S with Mode = Enc, such that R is known to the attacker, it must be that the peer has been compromised (leak(S_{pk})). Moreover, if the SecurityPolicy (*SPo*) is not RSA, and thus is ECC, then PFS requires that the leak must have occurred *before* the request ($t_s < t$). A similar property models the confidentiality from servers' point of view, using an event $S_{data}(\cdot)$ triggered whenever a request/response is received/sent. We respectively call those two properties $Conf_C$ and $Conf_S$.

$$\begin{aligned} & C_{data}(C_{pk}, S_{pk}, SPo, \mathsf{Enc}, S, R)@t \\ & \quad \wedge\ \mathsf{attacker}(R) \\ \Longrightarrow\ & \mathsf{leak}(S_{pk})@t_s \\ & \quad \wedge\ (SPo = \mathsf{RSA} \vee t_s < t) \end{aligned}$$

Figure 6.5.: Query $Conf_C$ modeling confidentiality of user requests, we omit the event(·) and quantifiers for readability.

$Conf_{Pwd}$: Confidentiality of passwords. A second confidentiality requirement is that passwords remain secret, see Figure 6.6. The property states that for any declared user, if the attacker knows their password, then one of the following must have happened: *(i)* the user has been compromised, or *(ii)* the password has been used by a client to activate a session (event CActivR(·)) with a compromised server S_{pk}. Indeed, in both cases the password is trivially leaked. Otherwise, we expect the password to remain confidential.

$$\begin{aligned} & \mathsf{User}(U, pwd, U_{sk}, U_{pk}, U_{cert}) \\ & \quad \wedge\ \mathsf{attacker}(pwd) \\ \Longrightarrow\ & \mathsf{leak}(U_{pk}) \\ & \quad \vee\ (\mathsf{CActivR}(C_{pk}, S_{pk}, m, S, U, pwd) \\ & \qquad \wedge\ \mathsf{leak}(S_{pk})) \end{aligned}$$

Figure 6.6.: Query $Conf_{Pwd}$ modeling confidentiality of passwords.

Agr_S and Agr_C: Data and entity authentication of user requests and responses. Finally, we model authentication of user requests. As discussed in Section 6.1.3, this property is an injective agreement property, see Figure 6.7. The property states that, whenever the server S_{pk} receives a request R from a user U (event SrcvR(·)), sent on a client C_{pk} in a session SE, then U did indeed send R to S_{pk} in session SE (event CsndR(·)), or the attacker can impersonate the user and either (a) the client was compromised (leak(C_{pk})) or (b) the channel mode is None. To impersonate the user we have to consider three cases: *(i)* either the session is anonymous, *i.e.,* there is no user, or *(ii)* the user credentials have been leaked, or *(iii)* the user did use their password to authenticate to a corrupted server (on an arbitrary client). By declaring the agreement to be injective, we ensure that any user request event on the server side corresponds to a *distinct* user request event on the client side. This is called Agr_S. The opposite property, Agr_C, is also modeled and starts with CrcvR(·), ends with a SsndR(·) event, and is simply conditioned by ∨ leak(S_{pk}).

$$\begin{aligned} & \mathsf{SrcvR}(C_{pk}, S_{pk}, SE, U, R) \\ \Longrightarrow\ & \mathsf{CsndR}(C_{pk}, S_{pk}, SE, U, R) \\ & \quad \vee\ \big(\mathsf{leak}(C_{pk}) \vee mode = \mathsf{None}\big) \\ & \qquad \wedge\ \big(U = \mathsf{anon} \vee \mathsf{leak}(U) \vee \\ & \qquad\quad (\mathsf{CActivR}(C'_{pk}, S'_{pk}, m, S', U, pwd) \\ & \qquad\qquad \wedge\ \mathsf{leak}(S'_{pk}))\big) \end{aligned}$$

Figure 6.7.: Injective query Agr_S modeling data and entity authentication of user requests, we omit the inj-event(·) for readability.

6.3. Analysis Results

We have leveraged ProVerif to analyze the properties stated in Section 6.2.2. Using ProVerif out of the box on the full model was not possible, due to the complexity of the model and the properties: the tool would not terminate or return false attacks. We had to design a careful proof methodology —discussed in Section 5.6— to be able to obtain meaningful results.

Our analysis uncovered several vulnerabilities affecting OPC UA version 1.05.03, some of them also affecting version 1.05.04. Each of these vulnerabilities, and the fixes we have designed to address them, have been responsibly disclosed to the OPC Foundation. Then, during the embargo

period, we discussed with the Foundation (through their ticketing system, emails, and attending their working group meetings) the different fixes and their impact on existing implementations. The Foundation decided to keep most of them, which are now part of the updated specification v1.05.07. As we shall explain in detail in Section 6.4, some vulnerabilities have not been entirely fixed (mostly for backward compatibility reason), some have been fixed much later (more than two years after our disclosure), and some other residual risks remain today. In the meantime, we had to weaken the properties Agr_S and Agr_C we could prove on the updated specification, to reflect the residual risks. Excluding those residual risks, we show that our fixes do indeed avoid these attacks in the configurations for which we initially found them.

We detail the found attacks, their impact, root causes and remediation in Table 6.2. We summarize our formal findings in Table 6.1.[3] For each of the vulnerabilities, we add links to tickets that describe follow-up actions and modifications to the standard resulting from our disclosure and the discussions with the OPC Foundation. Since the vulnerabilities and weaknesses we describe target a specification, rather than an implementation, we did not request vulnerability identifiers at first. The Foundation, however, decided to assign GCVE identifiers to the vulnerabilities we reported.[4]

3: The details of the maximal configurations for which we obtained proofs, as well as the configurations for which some residual risks remain, are detailed in [*DHK25a*, §5]. Finally, except for the signature oracle attack (Section 6.3.5), all vulnerabilities we report were automatically discovered using ProVerif on our model.

4: See Section 6.4.3 for more details on our responsible disclosure process and the acknowledgments by the OPC Foundation.

6.3.1. Client Impersonation in ECC

Our analysis uncovered that the supposedly more secure ECC policy suffers from a client impersonation attack when SessionSecurity = SNoAA that violates Agr_S and Agr_C. The attacker can deceive a server into believing that the user issuing the request is on a different client.

Description We illustrate the attack against Agr_S in Figure 6.8 and suppose an honest client C, a compromised client A (presented as the attacker) and an honest server S.

C sends an *OpenSecureChannel* request to S. The attacker intercepts this request and modifies the client's identity and resigns the request as coming from A. The server sends its response that the attacker forwards to C, who accepts and opens the channel. Indeed, when SecurityPolicy = ECC, the server's response does *not* include the client's identity. According to [OPC23, Part 6, § 6.7.2.3 table 51] the *ReceiverCertificateThumbprint*, that indicates what public key was used to encrypt the remaining of the

Properties	OPC UA v1.05.03		v1.05.04
$Conf_C$	✗	6.3.5	✓
$Conf_S$	✓		✓
$Conf_{Pwd}$	✗	6.3.5, 6.3.5	✓
Agr_S	✗	6.3.1, 6.3.2, 6.3.3, 6.3.4, 6.3.5	✗/✓$^-$
Agr_C	✗	6.3.1, 6.3.5, 6.3.5, 6.3.5	✗/✓$^-$

Table 6.1.: Summary of our automated security analysis with respect to the properties from Section 6.2.2. ✗: an or several attack(s) are automatically found by proverif, with references to the corresponding subsections (those in appendices are shown in brackets). ✓: proverif proved the property. ✗/✓$^-$: proverif shows that some residual risks remain in some configurations (attacks against Agr_*) and proves instead a weaker property that does not consider the residual risks (Agr_*^-). When we claim a proof: not all configurations have necessarily been proven, but all configuration options are covered (see [*DHK25a*, §5]).

message, *shall be null if the message is not encrypted*, which is the case in ECC (but not in RSA, for which the attack is impossible). Therefore, client C is unable to detect that the server's response was intended for A.

At this stage, this already violates authentication of the *OpenSecureChannel* request, which one could state as an intermediary property, as there is disagreement on the client identity between C and S. The attack is also reminiscent of Lowe's attack on the NSPK protocol [Low96].

We now show the attacker can continue and violate Agr_S. Once the channel is established, the client C will use it to create a session. A user on client C will activate this session and send a user request. These requests will be accepted when SessionSecurity = SNoAA as the server does *not* check that the client certificate corresponds to the one of the channel. Hence, the server believes that the request received stems form user U on client A while it actually originates from user U on client C. As illustrated by the events at the end of the client and server's execution in Figure 6.8, this disagreement violates Agr_S (note that user U is neither compromised nor using passwords). The same attack and disagreement on the client identity does actually also violate Agr_C.

Interestingly, this attack affects the newly introduced ECC but not the "old" and supposedly less secure RSA security policy since the *ReceiverCertificateThumbprint* is not null. It witnesses the fact that cryptographic suites cannot be simply replaced without a thorough security analysis.

Impact Such an attack could for instance be exploited to create wrong security logs or circumvent user authorization based on the client machine (*e.g.,* Role Based Access Control). Indeed, the server will associate the wrong client-user pair to a user request. A more dramatic example of a potential exploit scenario of the lack of agreement allows an attacker to redirect legitimate (say highly privileged) user U's requests at a training client C intended to a training server, to an in-production SCADA server S, who will accept and log them as coming from a control room client A, that we assume compromised (say decommissioned). This kind of confusion is made possible by the lack of agreement on the pair user-client and a "Non-Transparent Redundancy" server mode. We further explain this and report an additional, less serious Denial-of-service attack (DOS) attack in our full vulnerability report [*DHK25b*].

Mitigation and fix We proposed to fix this vulnerability by including *ReceiverCertificateThumbprint* (*i.e.,* hash of the receiver's public key) in ECC as well. This fix was accepted by the OPC Foundation and changes in the specification are documented in tickets #9349, #9350 and #9427 including the fix and several clarifications (notably the Conformance Unit 3781). In particular, they updated [OPC23, Part 6] v1.05.04 to *"require the ReceiverCertificateThumbprint to be set for all ECC policies"*. Since the issue is corrected in the specification, we included the fix in our model to show its effectiveness. In particular, while ProVerif finds an attack against $\overline{Agr_C}$ on v1.05.03 for configuration (ECC + Enc + SNoAA + Cert + Leak), it proves v1.05.04 secure for the same property and configuration.

Table 6.2. Summary of the attacks found that affect OPC UA v1.05.03. Additionally, all attacks except those that require ECC also affect OPC UA v1.04. Att(d) denotes the attacker should know or compromise d. We indicate with Agr_{CS} when both Agr_{C} and Agr_{C} are violated (similarly for the weakened variants). For agreement properties, we indicate in square bracket the piece of data on which the disagreement occurs. For instance, $Agr_{CS}[U]$ indicates that clients and servers both end up in disagreement on the user U who sent/received a request. See [*DHK25c*] for more information about the Table.We indicate in bold font the remediations chosen by the OPC UA Foundation to be included in the specification (prior to June 2025, see Section 6.4.3 for later changes) and in parentheses the tickets that can be found at https://mantis.opcfoundation.org/view.php?id=<ticket>.

Name (Section)	Violated Properties	Violation	Threat Assumptions	Configuration Assumptions	Practical Impact	Root Causes	Remediations: Fix, Mitigation (Mit), Enhancement (Enh) and their Tickets
Client Impersonation in ECC (6.3.1)	$Agr_{CS}[C]$	Server S receives a user request allegedly from C^a, U but it comes from C, U	Att(C^a_{sk})	ECC + SNoAA	Confusion (client,user) at server, potential confusion test vs prod. client, wrong client logging	• RC2: lack of receiver identity	• **Fix2: enforce receiver identity in ECC (9349)** • **Enh2: clarify stronger checks in SNoAA (9350, 9427)**
KCI: User Impersonation (6.3.2)	$Agr^{-0}_{S}[U]$	Server S receives a user request allegedly from C^a, U but user U is on client C and did not send it	Att(C^a_{sk}), Att(S_{sk})	Cert	User impersonation	• RC3: lack of binding of user's authentication (user's signature) to full context (client identity)	• Fix3: add client's identity (public key or certificate) in user's authentication signature • **Enh3: plan to release some enhanced security versions of the user tokens (9809, 9810)**
Session Hijack—Reopen (6.3.3)	$Agr^{-}_{S}[U]$	Server S receives a user request allegedly from C^a, U but user U is on client C and did not send it	Att(C_{sk})	Sign (+ Reopen)	User impersonation	• RC6: lack of binding btw channel symmetric keys before and after renewal (Reopen) • RC7: no session ownership proof (tok_s not secret)	• **Fix6: forbid reopen takeover by linking channel secret keys through renewal (10056)** • Fix7: keep tok_s secret even is Sign mode, and use it as a MAC computed on each request • **Mit2: disable Sign mode by default (9875)**
Session Hijack—Switch (6.3.3)	$Agr^{-}_{S}[U]$	Server S receives a user request allegedly from C^a, U but user U is on client C and did not send it	Att(C_{sk})	Sign (+ Switch)	User impersonation	• RC7: no session ownership proof (tok_s is not secret)	• Fix7: keep tok_s secret even is Sign mode, and use it as a MAC computed on each request • **Mit2: disable Sign mode by default (9875)**
KCI: Session and User Confusion (6.3.4)	$Agr^{-}_{S}[U]$	Server S receives a user request allegedly from C, U but U^d sent it and U^d can be a dummy user without any valid credential	Att(S_{sk})	(Reopen)	User impersonation	• RC6: lack of binding btw channel symmetric keys before and after renewal (Reopen) • RC7: no session ownership proof (tok_s not secret)	• **Fix6: forbid reopen takeover by linking channel secret keys through renewal (10056)** • Fix7: (keep tok_s secret even is Sign mode, and) use it as a MAC computed on each request
Race Condition for User Contexts (6.3.5)	$Agr_{CS}[U]$	Server S receives a user request allegedly from C, U but U_2 sent it	∅	∅	Abuse of user rights, wrong logging (user)	• RC1: missing binding btw user and activated session (SAToken tok_s)	• Fix1: bind tok_s to user (change at each activation) • **Mit1: limit new activation to a user with same or lower rights (9351)**
Downgrade of Password Secrecy (6.3.5)	$Conf_{Pwd}$	Compromission of user password	Att(ch_{sk})	Enc + Pwd	Stealing of user password, full user impersonation	• RC4: relaxed security configuration to avoid what may appear to be redundant encryption but is not	• Fix4: use a challenge response mechanism for user password authentication or use proper PAKE protocol • **Enh4: recommend dedicated password encryption, even in mode Enc (9432)**
Risk of Signature Oracle (6.3.5)	$Conf_{C}$, $Conf_{Pwd}$, Agr_{C}	Server impersonation towards honest clients and users in ECC, a less severe UKS attack for RSA	∅	ECC + SNoAA	Stealing of user password, full server impersonation	• RC5: lack of context in signatures	• Fix5: add context and tags to the signature • **Enh5: checked well-formedness of certificates (9594, 9596, 9597 and 9598) and nonce length (9595, 9599)**

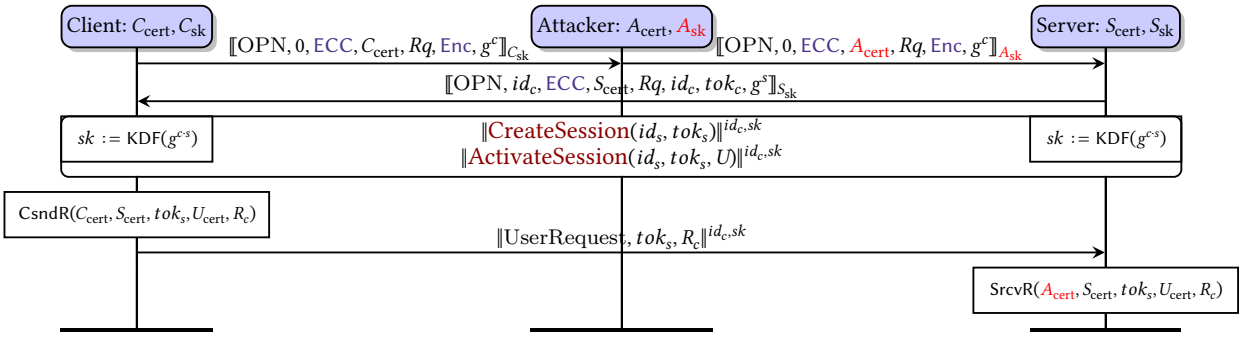


Figure 6.8.: Client impersonation attack when SecurityPolicy = ECC and SessionSecurity = SNoAA. Lack of agreement on the client during user Authentication is made possible by the same weakness that allows the KCI attack, *i.e.,* the lack of link to the client.

6.3.2. KCI: User Impersonation

While key compromise trivially allows an attacker to impersonate the compromised party, a Key Compromise Impersonation (KCI) attack allows them to impersonate an uncompromised party to the compromised party. Modern protocols are designed to resist such attacks. For instance, the TLS 1.3 specification explicitly lists resistance to KCI attacks as one of its goals [Res18, App. E] for certificate-based authentication. Yet, we have discovered a KCI attack that allows the attacker to impersonate an honest user on an honest client to a server whose keys have been leaked.

We note that password-based authentication can be violated in OPC UA as soon as a user connects to a server whose keys have been compromised since clients send user's passwords to the server. Certificate based authentication is supposed to resist this since it involves a signature by the user's private key (*e.g.,* computed on a smart card) of a nonce generated by the server. The attack we found shows the contrary: a user can be impersonated even when using a certificate to authenticate.

Description The attack involves an uncompromised client C, an uncompromised user U, and a compromised server S. We also suppose that the keys of another client C^A have been compromised. We explain the attack when SecurityPolicy = ECC and Mode = Enc. The detailed message flow is provided in Figure 6.9.

1. The attacker opens a channel between C^A and S and creates a session se_1 between C^A and the honest server S, in which S generates the nonce S_{nonce} (see Figure 6.4).
2. C sends an Open Channel Request to S that is intercepted by the attacker. Knowing S's long-term keys, the attacker forges and sends back to C a valid response. Hence, the attacker also knows the channel keys.
3. C sends a Create Session Request to S. The attacker intercepts it. Knowing S's long-term keys and the channel keys, the attacker can impersonate the server and forges a valid Response. Moreover, the attacker uses the same server nonce S_{nonce} as in sessions se_1. C records this session se_2 in its local state.
4. User U on client C sends an activation request for se_2 to S. The attacker intercepts it and extracts the signature $[\![S_{cert}|S_{nonce}]\!]_{U_{sk}}$ created by U to authenticate, where U_{sk} is the private key associated to U_{cert}.
5. The attacker activates session se_1 using U's signature $[\![S_{cert}|S_{nonce}]\!]_{U_{sk}}$.

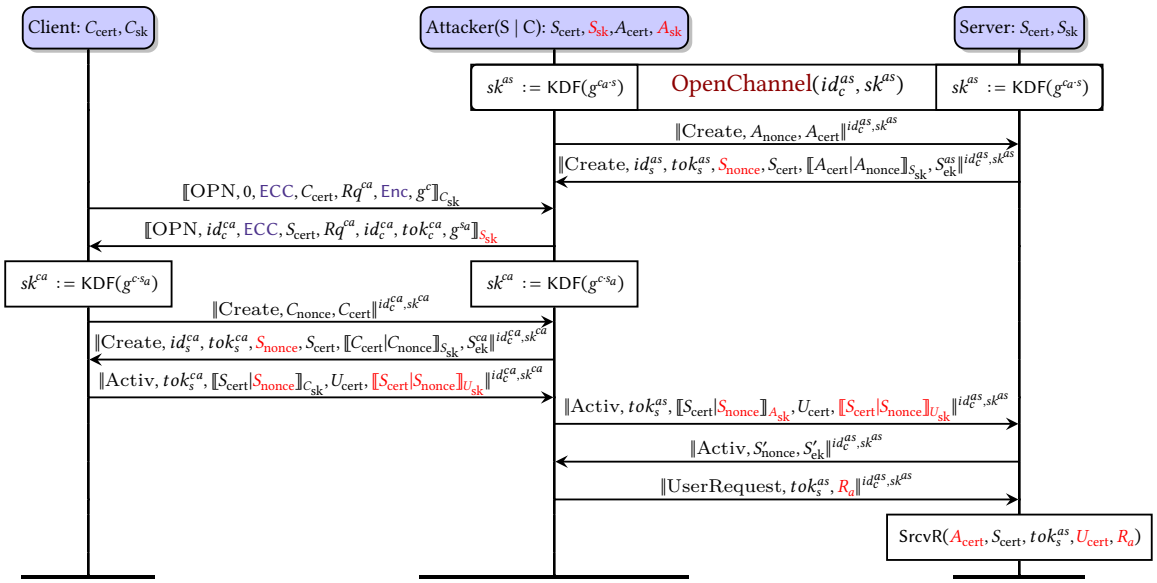


Figure 6.9.: KCI user impersonation attack when SessionSecurity = SSec (illustrated here when SecurityPolicy = ECC and Mode = Enc). Values in a session between the attacker and the server (respectively client) are annotated with ^{as} (respectively ^{ca}). We write OpenChannel(id_c^{as}, sk^{as}) for the channel open request and response messages with id_c^{as} identifier and keys sk^{as}.

As the server S has created S_{nonce} for session se_1, this session creation is accepted by S.

6. The attacker can now send arbitrary user requests in se_1 from client C^A without any further intervention from U. Note that the user U had no interaction with client C^A.

This attack violates Agr_S as a server accepts a user U's request without the user having sent any request.

Impact Assuming long-term keys S_{sk} of server S were compromised, an attacker can send arbitrary user requests to the legitimate server S on behalf of a user U, even if U authenticates with an uncompromised certificate on an uncompromised client (*e.g.,* using state-of-the-art authentication on a smart card), and even if U did not send any user request. We stress that we solely assume that S_{sk} are compromised (*e.g.,* through a flawed certificate management/renewal or flawed certificate storage/generation) but do not assume that the server is fully compromised. In particular, the server S could be protected with additional layers of security and can have access to high-privileged data and actions that the attacker does not have access to.

Mitigation and fix We proposed to fix this vulnerability by adding the client's identity to the user's signature, *i.e.,* $[\![C_{pk}|S_{cert}|S_{nonce}]\!]_{U_{sk}}$, where C_{pk} is the public key of the client's certificate. Such a fix would however break backwards compatibility of systems in production. The OPC Foundation acknowledged the attack, but considered the compromise of server long-term keys to be a very strong threat model that should be accepted for legacy systems, given the cost of a non-backward compatible fix.

To further analyze OPC UA, we had to weaken Agr_S^- to accept this residual risk, and to not deem as attacks the executions where a client is compromised and an honest user activates a session towards a compromised server (details on how this is formalizes can be found in [*DHK25c*]).

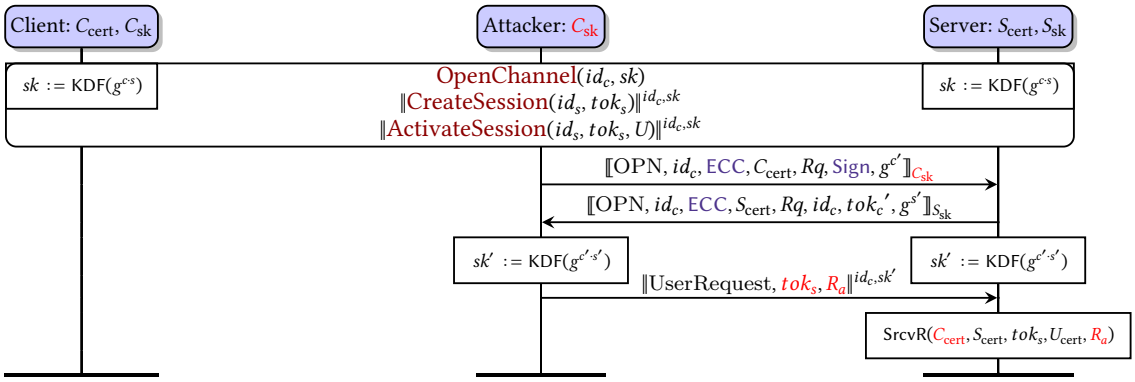


Figure 6.10.: Session hijacking by reopening a channel when Mode = Sign, illustrated here with SecurityPolicy = ECC and SessionSecurity = SSec.

A year after our responsible disclosure, the OPC Foundation has proposed and implemented a fix to this attack in v1.05.07; see Section 6.4.2. We plan to analyze this new version in future work.

6.3.3. Session Hijack by Reopening or Switching

Even though the weakened property Agr_S^- purposely avoids the previous attacks, we were able to uncover yet another attack when channels only provide integrity, *i.e.*, Mode = Sign.

Description The attack involves a compromised client C, an uncompromised user U and an uncompromised server S. The detailed message flow is provided in Figure 6.10.

1. Client C (whose key C_{sk} has been compromised) opens a channel in Mode = Sign and creates a session with S. The attacker learns the session token tok_s as only integrity is provided. (Note that although the attacker knows C_{sk}, it does not know the channel keys sk.)
2. Honest user U on client C activates the session using its user credentials (which are not leaked to the attacker).
3. Using C_{sk} the attacker re-opens the channel with S and learns the updated channel keys sk'.
4. Knowing sk' and tok_s, the attacker can send arbitrary user requests impersonating U.

Session Hijack by Switching variant Note that a variant of this attack relies on switching the channel of a session, rather than reopening the channel. In the above attack, a channel reopening request was issued by the attacker to gain access to the channel underlying the session. Alternatively, an attacker can intercept a session re-activation, open a new channel, and replay the re-activation attack on the new (attacker controlled) channel (see [*DHK25c*, §C.2] for details).

Impact Assuming that a client's long-term is compromised, an attacker can completely hijack the session of a user who activated a session on that client. The attacker can therefore send arbitrary requests on behalf of U.

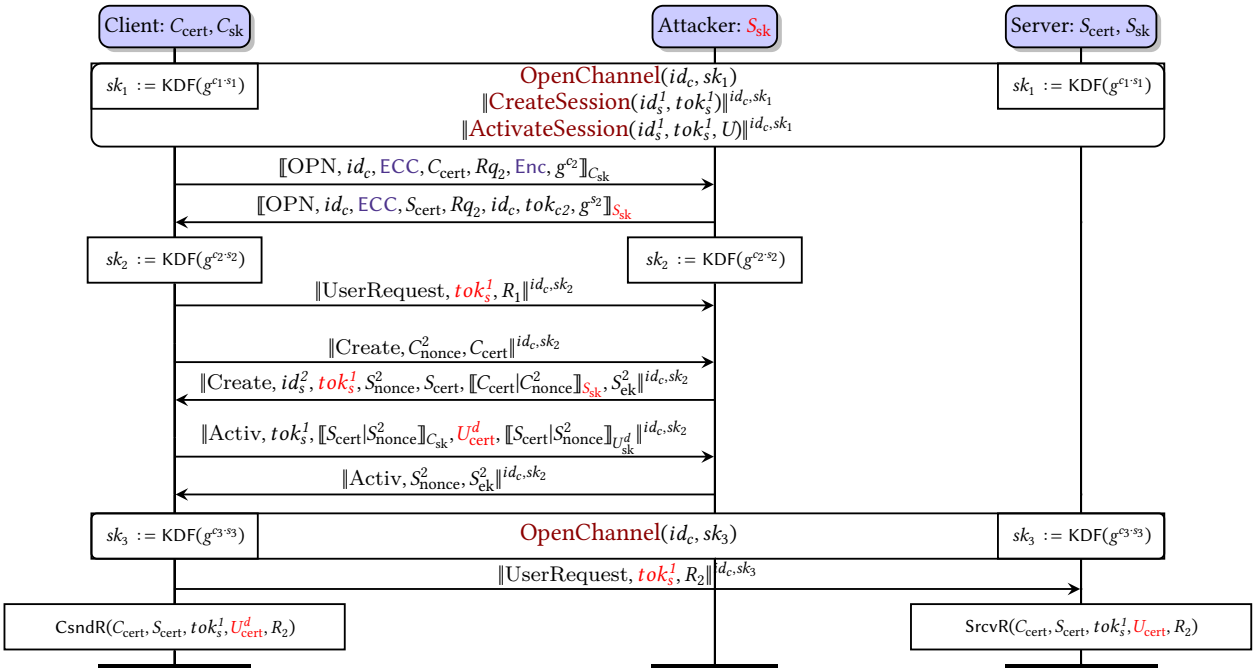


Figure 6.11.: Session confusion illustrated here with Mode = Enc, SecurityPolicy = ECC and SessionSecurity = SSec.

Mitigation This attack has also been acknowledged by the OPC Foundation. As a short-term counter-measure, the foundation has considered deactivating the mode Mode = Sign by default (tickets #9874, #9875).

For a more robust, long-term fix, the foundation has later proposed a mechanism where channel keys are chained through reopening (ticket #10056), which requires knowledge of previous channel keys to initiate a Reopen. This fix has been implemented in version 1.05.07 (see Section 6.4.2), which we plan to analyze in future work.

6.3.4. KCI: Session and User Confusion

As a response to our Session Hijack attacks, the OPC Foundation had first decided to disable Mode = Sign by default. However, even when Mode = Enc, we found another KCI attack that violates the weakened property Agr_S^-, which excluded the previous KCI attack (Section 6.3.2). We shall also assume a compromise of a server's long-term key. Recall that an impersonation of this server is obviously possible. What we additionally show is an impersonation of honest users towards that honest server.

Description The attack involves an honest client C and user U, a compromised server S. We additionally assume the adversary has normal access to C but is not necessarily registered in the user database. The detailed message flow is provided in Figure 6.11.

1. User U on client C opens a channel, creates and activate a session s_1 with server S.
2. When C renews the channel keys (Reopen), the attacker impersonates S and learns the new channel keys sk_2. The attacker will then learn tok_s^1 for session s_1.
3. The adversary initiates a new session s_2 creation and activation on C with a *dummy user* U^d (U^d does not have to be a valid, registered

user). The adversary continues impersonating S: it proceeds and accepts those requests and uses the exact same tok_s^1 from s_1. C has now registered two sessions s_1 (with U) and s_2 (with U^d), both with the same tok_s^1. Note that S is neither aware of session s_2 nor of user U^d.
4. C will eventually reopen the channel with the honest server S: the attacker can pretend to lose the connection, or can wait for the channel to be reopened or switched.
5. From now on, when C sends requests from U^d to S in session s_2, it uses tok_s^1, which causes S to believe this request is coming from the honest user U in session s_1.

Impact The impact is similar to the previous KCI attack (Section 6.3.2). Assuming long-term keys of server S were compromised, an attacker having unprivileged access to an honest client C can send arbitrary user requests to the legitimate server S on behalf of an honest, possibly high-privileged, user U, who previously activated a session on that client. Without assuming having access to C, a variant of this attack where U^d is a valid, honest user is still possible and has a similar impact as the Race Condition attack (see below): confusion of user context for user requests. More precisely, user requests are attributed to the wrong user from the server's point of view. Therefore, a user with less security clearance or privileges than another could then abuse the access rights and possibly exploit this for privilege escalation. It also leads to wrong security logs and user requests attributions. Note that the attack is also possible when Mode = Sign.

Mitigation This attack has also been acknowledged by the OPC Foundation, who eventually decided on fixing it by chaining channel keys through reopening (ticket #10056); see Section 6.4.2.

6.3.5. Other Findings

We report on additional attacks and weaknesses in [*DHK25c*] that we briefly summarize here.

Race Condition for User Contexts We discovered a *race condition* after the handover of a session to a new user, that breaks the authentication of user requests, formally both Agr_C and Agr_S. This violation occurs on all versions of OPC UA and does not require any key compromise. Suppose that an honest client creates a session, that is activated by user U^1. Later, the session is transferred to user U^2 by a new activation request and roughly at the same time user U^1 sends a user request R. Because the client has not received yet the server's activation response confirming the handover to user U^2, the client sends R in the context of user U^1 while the server is deceived into believing that the session is now associated to user U^2. This is only possible if a user request gets processed after a client initiates an activate session but before it receives the server's activate session response. This on-client race-condition pre-requisite does not seem forbidden by the specification and ultimately depends on client implementations. The attack actually relies on the fact that a user handover neither modifies the session identifier nor the session token, none of them being bound to the user identity.

The confusion of user context could be avoided by using a fresh *Session Authentication Token* tok_s for each new user, *i.e.,* in each session activation

response ([OPC23, Part 4, § 5.6.3.2 table 17]). This would avoid the above attack, but would not be backward compatible.

The OPC Foundation has acknowledged the problem, and plans to include the following mitigations in v.1.05.04 (#9351): *(i)* clients should *"create a new session for the new credentials, do the higher privilege operation and close the sessions"*; *(ii) "do not process new requests until activate session completes; any existing requests finish with the current credentials"*. As a result, the privilege escalation exploit discussed above is no longer possible. However, Agr_S and Agr_C are still formally violated due to the lack of agreement on the user's identity (and not only their rights). Therefore, in order to further analyze Agr_S and Agr_C, we had to weaken the properties to allow the user to differ, if there has been a previous activation with another user. The weakened properties are formally defined in [*DHK25c*, §A]. Another concern with the above mitigation is that it requires that users taking over activated sessions must have the same privileges as the user who previously activated the session. However, this approach seems impractical as it prohibits session takeovers by supervisors or administrators, a common and necessary use case in industrial systems.

Downgrade of Password Secrecy Our analysis uncovered that, when channel keys are leaked, counter-intuitively, Mode = Enc is less secure than Mode = Sign since it leaks user passwords. This is due to a relaxed security requirement in the specification that allows passwords to be sent in plaintext when Mode = Enc since they are under channel encryption anyway. Channel keys can be considered less secure than long-term keys in some scenarios (*e.g.,* when stored in memory while long-terms keys are stored in Hardware Security Module (HSM)s). The OPC Foundation acknowledged this issue and decided to add a warning and a recommendation to always encrypt passwords (see ticket #9432).

Risk of Signature Oracle A server configured to allow SNoAA may not verify the certificate sent by a client application at session creation, but simply append the client nonce and sign the resulting bitstring, to prove possession of its private key. This induces a "signature oracle" to produce valid server signatures on adversarially-chosen data. In particular, this signature oracle can be exploited to forge valid *OpenChannel* response and *CreateSession* responses, allowing the attacker to fully impersonate an honest server towards an honest client in ECC. Hence, all security goals are violated and the attacker can learn users' passwords. For RSA, the attack is less severe as the attacker does not learn the channel keys and can only perform an Uknown Key-Share (UKS) attack.

The OPC Foundation acknowledged the weakness, and to mitigate the vulnerability, *(i)* adds explicit requirements for v.1.05.04, and conformance tests to ensure that certificates are validated before being used to produce a signature (tickets #9594, #9596, #9597 and #9598); *(ii)* states that all Nonces shall have a length specified by the SecurityPolicy (tickets #9595, #9599). We reflect those fixes in our model and prove that they thwart the attack we found on the same configuration.

Interestingly, some attacks against OPC UA using the same signature oracle as one of the attack vectors were presented at Black-Hat [Ter25] and DEF CON [Ter25] in August 2025 and that were responsibly disclosed in February 2025 after our eprint [*DHK25a*] was made public and almost 1 year after our own responsible disclosure to the foundation.

6.4. Lessons Learned and Discussions with the OPC Foundation

We now step back and discuss the lessons learned from our analysis of OPC UA and the interactions with the OPC Foundation that followed our disclosure of the attacks.

6.4.1. Key Lessons on Protocol Design

Our analysis highlights that many attacks exploit *(i)* the multi-layer nature of the protocol: channels, created and activated sessions, requests, *(ii)* the agility of the protocol due to OT constraints: switch/reopen channels, re-activate sessions, etc., *(iii)* the complexity of the protocol: combination of multiple options and configurations. We believe that this protocol complexity is not inherently insecure, but requires a careful analysis. Several protocol design flaws are the root causes of our attacks and are often related to missing bindings between critical components, such as (see Table 6.2 for a detailed list): activated sessions are not bound to their user (6.3.5), signed requests are not bound to the recipient (6.3.1), signed user tokens are not bound to the client (6.3.2), new channel keys obtained by reopening are not bound to previous keys (6.3.3, [*DHK25c*], 6.3.4), and the absence of a context binding in signatures (6.3.5). This should be remembered when designing future protocol evolutions.

Our analysis also highlights the pitfalls of adding new features on top of an existing protocol. For instance, the addition of ECC by simply replacing a cryptographic primitive suite by another one has led to a quite subtle attack (6.3.1).

6.4.2. Fixes of the Main Remaining Attacks in v1.05.07

After the publication of the first version of our paper [*DHK25a*], the OPC Foundation has informed us in June 2025 that some of the attacks described in Table 6.2 should be fixed in the OPC UA v1.05.07 RC specification [OPC25] and asked us to review the proposed fixes. All attacks of Table 6.2 are supposed to be fixed in this new version for *SecurityPolicies* that require a new *SecureChannelEnhancements* policy, except the weakness Race Condition for User Contexts (6.3.5) which would still affect the protocol. For the latter, we have already discussed a proper fix *Fix1: bind tok_s to user (change at each activation)* in Section 6.3.5. We have also discussed a mitigation *Mit1: limit new activation to a user with the same or lower rights (9531)*, although it has severe practical drawbacks as explained at the end of Section 6.3.5.

A summary of the main changes, according to [OPC25, Part 6,§ 6.1 *"Security Handshake and Security Policies, Table 48"*] *"SecureChannelEnhancements"*, is as follows:

- *(i)* chaining the derivation of symmetric keys when reopening secure channels,
- *(ii)* adding session transfer tokens in activations used to switch between secure channels, and
- *(iii)* using channel-bound and client-certificate-bound signatures at session creations and activations, *i.e.,* in user's signatures.

The new fixes proposed by the OPC Foundation are only activated by new *SecurityPolicies* that enable *SecureChannelEnhancements*, but do not

address existing *SecurityPolicies*. We stress that they have not yet been verified by formal proofs. We strongly recommend to verify the security of the new *SecureChannelEnhancements* with such formal proofs, once the new designs are considered stable enough. We have not done this ourselves yet given the always evolving nature of the specification during our responsible disclosure process and the high opacity of their design process (see Section 6.4.3). A summary of our suggestions and warnings is as follows:

1. First of all, the weakness "Race Condition for User Contexts" (6.3.5) is not addressed, although *Fix1: bind tok_s to user (change at each activation)* could be included by providing a new tok_s in each activation response when required by the SecurityPolicy.
2. Otherwise, the proposed fixes seem appropriate *for the new policies that conform to SecureChannelEnhancements.*

6.4.3. Responsible Disclosure and Actions Taken by the OPC Foundation

We initiated our first responsible disclosure in October 2023 with the OPC Foundation, reporting all the vulnerabilities we found in the OPC UA protocol. After a long period of discussions, the Foundation has committed to addressing the vulnerabilities in June 2025 for the next OPC UA v1.05.07 RC specification, which was later released in December 2025. The OPC UA Foundation requested our review of the fixes that they planned to implement. We have summarized them in Section 6.4.2 and provide next a timeline for our responsible disclosure.

The vulnerabilities were reported to the OPC Foundation in a responsible manner through tickets on https://mantis.opcfoundation.org/ and our vulnerability report [*DHK*], before publishing our paper [*DHK25c*].

For each responsible disclosure, we engaged in embargo periods of at least 2 months to allow the OPC Foundation to fix the vulnerabilities before public disclosure. The foundation has generally been responsive to the reported tickets, engaging in multiple discussions and meetings with us to address the vulnerabilities and explore potential remediation plans.

However, the foundation has frequently pushed back: expressing concerns that our proposed fixes might compromise backward compatibility with earlier protocol versions, and asserting that the vulnerabilities are unlikely to occur in practice. Instead, the foundation has often suggested under-specified long-term fixes while offering minimal short-term mitigations. This approach has left several vulnerabilities unresolved, even months after being reported, which we found concerning and inadequate. Finally, after two meetings with the OPC Foundation Security Working Group, the German BSI, the American CISA and FBI, and the French ANSSI in May 2025, the foundation has committed to addressing the vulnerabilities in the next OPC UA v1.05.07 RC specification, which was eventually released in December 2025. In 2026, the foundation assigned GCVE identifiers to the attacks we found.

The timeline of the disclosure is as follows (we highlight in *italics and purple* (resp. *and red*) the items related to the vulnerabilities that were still unresolved prior to v1.05.07 in June 2025 (resp. for v1.05.07)):

2023 – October 10th Responsible disclosure of the "Client impersonation in ECC" attack (Section 6.3.1) on Mantis, ticket 9190 and report [*DHK*, §A]. *This has been fixed in v1.05.04 in October 2024.*

2023 – October 10th Responsible disclosure of the "Race Condition for

User Contexts" attack (Section 6.3.5) on Mantis, ticket 9191 and report [*DHK*, §B]. *A mitigation, which does not entirely fix the problem, has been proposed in the ticket 9351. Until June 2025 (after 1.5 years), the vulnerability has not been properly fixed but only mitigated with a recommendation that seems unrealistic in practice (see Section 6.3.5). As of today, there is no clear remediation plan in the upcoming version (v1.05.07).*

2024 – February 16th Responsible disclosure of the vulnerability "Downgrade of Password Secrecy" (Section 6.3.5) on Mantis, ticket 9432, which is still flagged as "Open" as of today, and report [*DHK*, §C]. *A warning and a recommendation to always encrypt passwords have been proposed, but are still to be added to the specification.*

2024 – June 7th Responsible disclosure of the vulnerability "Risk of Signature Oracle" (Section 6.3.5) on Mantis, ticket 9577 and report [*DHK*, §D]. Requirements are added, to ensure that: *(i)* certificates are validated before being used to produce signatures (#9594, #9596, #9597 and #9598); *(ii)* nonces have a length specified by the SecurityPolicy (#9595 and #9599). *We consider this has been appropriately fixed in v1.05.04 in October 2024.*

2024 – August 10th Responsible disclosure of "KCI: User Impersonation" attack (Section 6.3.2) on Mantis, ticket 9762 and report [*DHK*, §E]. *For detailed discussions on Mantis on possible mitigations and fixes since September 2024, see: (i) add CertificateBound JWTs to the list of possible JWT fields* (#9809), *(ii) explain how risk of UserToken interception can be mitigated with CertificateBound tokens* (#9810), *(iii) add a ChannelThumbprint Identifier to all secure channels* (#10055), *Later (2024, September 3rd), the foundation "plan to deprecate the less secure modes (user certificate signing) in favour of more secure modes (OAuth2/JWTs)". While those new potential authentication modes seem to improve the security of the protocol, they do not address the root cause of the vulnerability. Until June 2025 (after 9 months), the considered remediation plan is under-specified and is foreseen in the mid-term, and this vulnerability remains unaddressed in the latest version of the OPC UA protocol (v1.05.04). This has been later fixed in v1.05.07 in December 2025 for the new security policies that conform to SecureChannelEnhancements (see Section 6.4.2). As of today, there is no remediation plan for policies that do not conform to SecureChannelEnhancements.*

2024 – March 21st Agreed to fix the "Client impersonation in ECC" attack in 1.05.04 RC1, see ticket 9349; 5 months after disclosure. *We consider this has been appropriately fixed in v1.05.04 in October 2024.*

2024 – October 7th Responsible disclosure of the "Session hijacking by reopening" attack (Section 6.3.3) on Mantis, ticket 9871 and report [*DHK*, §F]. *For detailed discussions on Mantis on possible mitigations, see: (i) addressing ARP poisoning attacks* (#9874), *(ii) disabling Sign only mode* (#9875), *(iii) chaining channel symmetric keys through reopening* (#10056), *(iv) forbiding channel reopening* (new message in #10056). *Until June 2025 (after 7.5 months), the considered remediation plan is under-specified and is foreseen in the mid-term, and this vulnerability remains unaddressed in the latest version of the OPC UA protocol (v1.05). This has later been fixed in v1.05.07 in December 2025 for the new security policies that conform to SecureChannelEnhancements (see Section 6.4.2). As of today, there is no remediation plan for policies that do not conform to SecureChannelEnhancements.*

2024 – November 25th Responsible disclosure of the "Session confusion due to a Key Compromise Impersonation attack" (Section 6.3.4) on Mantis, ticket 10035 and report [*DHK*, §G]. *Some mitigations are*

only mentioned on Mantis, such as chaining channel symmetric keys through reopening. Until June 2025 (after 6 months), the considered remediation plan is under-specified and is foreseen in the mid-term, and this vulnerability remains unaddressed in the latest version of the OPC UA protocol (v1.05) This has later been fixed in v1.05.07 in December 2025 for the new security policies that conform to SecureChannelEnhancements (see Section 6.4.2). As of today, there is no remediation plan for policies that do not conform to SecureChannelEnhancements.

2024 – November 26th Responsible disclosure of the "Session hijacking by switching" attack ([*DHK25a*, §C.2]) on Mantis with a new message added to the ticket 9871 and report [*DHK*, §F]. *Until June 2025 (after 6 months), the considered remediation plan is under-specified and is foreseen in the mid-term, and this vulnerability remains unaddressed in the latest version of the OPC UA protocol (v1.05). This has later been fixed in v1.05.07 RC in December 2025 for the new security policies that conform to SecureChannelEnhancements (see Section 6.4.2). As of today, there is no remediation plan for the policies that do not conform to SecureChannelEnhancements.*

2025 – January 30th Public ePrint of the paper [*DHK25c*].

2025 – May 14th Our paper is officially published at USENIX Security [*DHK25a*] and presented at the corresponding symposium in August 2025.

2025 – May Two meetings with the American CISA and FBI, German BSI, French ANSSI, as well as the OPC Foundation.

2025 – June The OPC Foundation sent us the fixes it plans to implement in the next OPC UA v1.05.07 RC specification, which was expected to be released in 2025 at the time, and asks for reviews and feedback; see Section 6.4.2 for our response. Clarification of the fixes later sent by email.

2025 – August Presentation of the paper [*DHK25a*] at USENIX Security.

2025 – December Release of OPC UA v1.05.07 RC.

2026 – February The OPC Foundation sent us a first draft of the 8 GCVE entries it had planned to file for the attacks we disclosed.

2026 – April Release of OPC UA v1.05.07.

2026 – August After further discussions with the OPC Foundation regarding these entries between February and August, it eventually finalized them.[5]

5: https://github.com/OPCFoundation/OPC-SecurityAdvisories

6.5. Conclusion

We proposed a verification framework for OPC UA based on a comprehensive protocol description, a ProVerif model using a number of advanced ProVerif features and a new proof methodology, and some tooling to mitigate the complexity of the proof effort (see Chapter 5). Our framework automatically discovered several vulnerabilities that have all been acknowledged by the OPC Foundation. In response, the specification has been updated with fixes and mitigations.

We believe that our framework can serve as a starting point for other researchers and practitioners working with OPC UA. First of all, our proof methodology may also serve for other ambitious case studies whose automated verification do not work "out-of-the-box". Second, our formal models can be reused and extended for further analysis of OPC UA, following future evolutions of the standard. As discussed, the protocol residual risks induced additional analysis challenges, especially for proving the agreement properties. Should those risks be accepted by the Foundation

in the longer term, an interesting future work is to improve our proof methodology and develop dedicated invariants to obtain complete proofs for maximal configurations. Moreover, once the OPC UA Foundation eventually decides on a stabilized list of fixes and mitigations, we strongly recommend to formally verify they indeed address the vulnerabilities we found and do not introduce other issues.

We also believe that the lessons we drew from this analysis (Section 6.4.1) can inform future design choices of ICS protocols. More broadly, we advocate for stronger collaboration between standardization bodies such as the OPC UA Foundation and the research community. We believe such bodies could better incentivize academic scrutiny by improving their responsible disclosure processes (see Section 6.4.3) and increasing transparency in their response mechanisms—for instance, by keeping disclosers informed of specification updates, providing public acknowledgment of contributions, and so forth.

Challenging and Improving the 4G and 5G Mobile Communication Standards

7.

New Practical Privacy Attacks [Bor+19; OBH17] and Formal Analysis [Bas+18]

7.1 Simplified Overview of 5G AKA 100
7.1.1 Architecture 101
7.1.2 Authentication Protocols 101
7.2 Formal Analysis of 5G AKA 104
7.2.1 Summary of our 5G AKA Analysis 105
7.2.2 Selection of Key Results of our 5G AKA Analysis . . . 106
7.3 New Privacy Risks and Practical Attacks . . . 109
7.3.1 Privacy Risks over WiFi . 109
7.3.2 Privacy Risks of AKA in 3G, 4G, and 5G 109
7.4 Lessons Learned and Discussions with GSMA and 3GPP 115
7.4.1 Recommendations for the 5G AKA Standard 115
7.4.2 Recommendations Regarding Privacy 116
7.4.3 Challenges with Formal Analysis of Privacy Properties for AKA Protocols . . 117
7.4.4 Responsible Disclosure and Further Impact 118

Roughly 70% of the world's population, *i.e.,* well over 5 billion people, are mobile subscribers [GSM17; GSM26]. They are connected to the mobile network via their USIM cards and are protected by security mechanisms standardized by the 3rd Generation Partnership Project (3GPP) group. Both subscribers and carriers expect security guarantees from the mechanisms used, such as the confidentiality of user data (*e.g.,* voice and SMS) and that subscribers are billed only for the services they consume. Moreover, these properties should hold in an adversarial environment with malicious base stations and users.

One of the most important security mechanisms in place aims at mutually authenticating subscribers and their carriers and establishing a secure channel to protect subsequent communication. For network generations (3G and 4G) introduced since the year 2000, this is achieved using variants of the Authentication and Key Agreement (AKA) protocol, standardized by the 3GPP. These protocols involve the subscribers, the Serving Networks (*SNs*) that have base stations in subscribers' vicinity, and Home Networks (*HNs*) that correspond to the subscribers' carriers. The protocols aim to enable the subscribers and the *HNs* to mutually authenticate each other and to let the subscribers and the *SNs* establish a session key. In a nutshell, AKA protocols are challenge-response protocols mainly based on symmetric cryptography and a *sequence number* (SQN) to verify the freshness of challenges, preventing replay attacks.

Next-Generation (5G) Since 2016, the 3GPP group has been standardizing the next generation of mobile communication (5G) with the aim of increasing network throughput and offering an ambitious infrastructure encompassing new use cases. The 5G standard has been deployed in two phases. The first phase (Release 15, June 2018) addressed the most critical requirements needed for commercial deployment and formed the basis for the first deployment. The second phase (Release 16, July 2020) addressed all remaining requirements.

In June 2018, the 3GPP published the *final* version v15.1.0 of Release 15 of the Technical Specification (TS) defining the 5G security architecture and procedures [3GP18]. The authentication in 5G Release 15 is based on new versions of the AKA protocols, notably the new *5G AKA* protocol, which enhances the AKA protocol used in 4G (EPS AKA) and which supposedly provides improved security guarantees. Reading early drafts of the 5G standard in 2017, we asked ourselves:

> *What are the security guarantees that 5G AKA actually provides and under which threat model and security assumptions?*

Privacy of AKA While privacy was an explicit requirement for 3G and 4G [3GPa; 3GPb], numerous *fake base station attacks* have been shown to compromise subscriber privacy in these networks [Ara+12; Hah+14; Sha+16; UB; BVR15; KM14; ZF05; KAC08; BS13]. The fake base

station attacks typically exploit weaknesses in the AKA protocol such as the non-protected identity request mechanism (*e.g.,* with so-called IMSI-catchers [Sha+16; UB; BVR15; KM14; ZF05; KAC08; BS13]) and the privacy-leak resulting from authentication failure messages [Ara+12; Hah+14]. In practice, those attacks break *subscriber location privacy*: an attacker can identify if certain subscribers are in the range of attacker's fake base stations. For 5G, the 3GPP has improved AKA in order to mitigate these well-known privacy issues [3GPc]. The 5G AKA protocol notably introduces randomized asymmetric encryption for protecting identifiers sent prior to authentication. Natural questions that we asked ourselves at the time were the following:

> *Is this countermeasure sufficient to protect subscriber location privacy in 5G AKA? Are there other privacy risks caused by the AKA protocol in 5G but also in earlier generations?*

Privacy Risks over WiFi In addition, AKA is also used in Extensible Authentication Protocol (EAP) mechanisms (*e.g.,* EAP-AKA, EPS-AKA, EPS-AKA') to secure point-to-point protocol authentication methods, wireless LAN internetworking, and generic authentication architectures including generic solutions for securing HTTP based services [AH06; Pet]. In particular, one key use case for EAP-AKA is for smartphones to connect to mobile services through WiFi: WiFi-Calling or Internet services. To improve usability, the connection to, and use of, WiFi services has been automated as much as possible, in particular the mutual authentication between the phone and the service provider through protocols like EAP-AKA. This opens up mobile phones to a new attack vector and channel by sniffing and intercepting messages on the WiFi layer. Note that such an attacker is more realistic than the *fake base station attacker*, which requires costlier hardware and activity on a protected wireless spectrum. We hence asked ourselves:

> *Do smartphones operating over WiFi expose subscribers to new privacy risks?*

In three of our prior works [*OBH17*; *Bor+19*; *Bas+18*], we have addressed these research questions. After a brief overview of the 5G AKA protocol in Section 7.1, we first present our formal analysis of 5G AKA in Section 7.2 revealing flaws and imprecisions in the 5G specification. Next, we present our new subscriber location privacy attack over WiFi as well as a new, more severe, kind of privacy attack against 3G, 4G, and 5G AKA protocols in Section 7.3. While these attacks were found through manual inspections of the protocol standards, we used formal methods to assess the validity of fixes. We focus here on the attacks and their impact. Finally, we conclude in Section 7.4 with lessons learned, explicit recommendations and fixes to improve the standards and the protocols as well as a summary of our responsible disclosure process with 3GPP, GSMA, and some impacted vendors.

7.1. Simplified Overview of 5G AKA

We explain in this section how authentication and key establishment are achieved in the 5G ecosystem, following as closely as possible the specification 3GPP TS 33.501 [3GP18], referred from here on as [TS 33.501]. We simplify the terminology and simplify the protocol to improve readability.

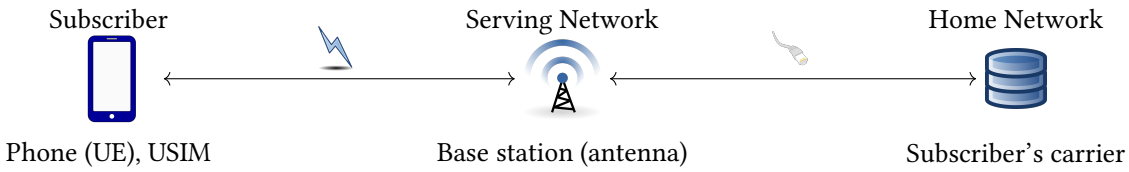


Figure 7.1.: Overall architecture in mobile communications: The subscriber uses their phone (UE), equipped with a USIM, to communicate with a base station run by the *SN* over an insecure wireless channel. The *SN* communicates with the subscriber's carrier (*HN*) on the right over an authenticated (wired) channel.

A detailed and formal description of the protocols can be found in [*Bas+18*]. We first present the general architecture and afterwards the authentication protocols, notably 5G AKA. We finally summarize the key differences of earlier generation AKA protocols (3G and 4G) with respect to 5G AKA.

7.1.1. Architecture

Three main entities are involved in the cellular network architecture (see Figure 7.1). First, *User Equipment* (*UE*), typically smartphones or IoT devices containing a *Universal Subscriber Identity Module* (USIM), are carried by *subscribers*. We shall call a *subscriber* the combination of a *UE* with its USIM. Second, *Home Networks* (*HNs*) contain a database of their subscribers and are responsible for their authentication. However, subscribers may be in locations where their corresponding *HN* has no *base station* (*i.e.*, antennas which may connect *UEs* to the network), for example when roaming. Therefore, the architecture has a third entity: the *Serving Networks* (*SNs*) to which *UEs* may attach to. An *SN* provides services (*e.g.*, call or SMS) once both the *UE* and the *SN* have mutually authenticated each other (this supports billing) and have established a secure channel with the help of the subscriber's *HN*. The *UE* and *SN* communicate over the air, while the *SN* and *HN* communicate over an authenticated channel (we list security assumptions later in this section).

As mentioned earlier, each subscriber has a USIM with cryptographic capabilities (*e.g.*, symmetric encryption, MAC). Relevant for our work is that the USIM stores:

- a unique and permanent subscriber *identity*, called the *Subscription Permanent Identifier* (*SUPI*),
- the public asymmetric key pk_{HN} of its corresponding *HN*,
- a long-term *symmetric key*, denoted as K (used as a shared secret between subscribers and their corresponding *HNs*), and
- a counter, called *Sequence Number*, denoted as *SQN*.

The *HN*, associated to some subscriber, stores the same information in its database.

7.1.2. Authentication Protocols

To enable *SNs* and subscribers to establish secure channels and authenticate each other, the 3GPP has specified two authentication methods: *5G AKA* and *EAP-AKA'*. We next describe 5G AKA, see [*Bas+18*] for more details about other variants.

Initialization Protocol [TS 33.501, Sec. 6.1.2] Figure 7.2 depicts the sub-protocol responsible for the subscribers' identification and initializing

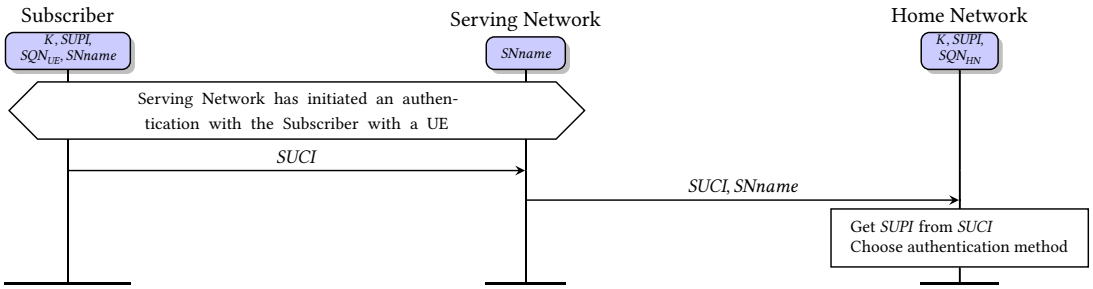


Figure 7.2.: Initiation of Authentication in 5G AKA. *SUCI* encrypts *SUPI* with *HN*'s public key to protect subscriber's identity.

the authentication. Once the *SN* has triggered an authentication with the subscriber, the latter sends a randomized encryption of the *SUPI*: $SUCI = \langle \text{aenc}(\langle SUPI, R_s \rangle, pk_{\text{HN}}), idHN \rangle$, where aenc(·) denotes asymmetric encryption, R_s is a random nonce, and *idHN* uniquely identifies an *HN*. This is a major improvement over earlier generations (3G and 4G), where *SUPI* (called *IMSI* in those generations) was sent in clear text over the air interface, thus allowing passive attackers to trivially track subscribers (so-called *IMSI catchers*) The identifier *idHN* enables the *SN* to request authentication material from the appropriate *HN*. Upon reception of the *SUCI* along with the *SN*'s identity (referred to as *SNname*), the *HN* can retrieve the *SUPI*, the subscribers' identity, and choose an authentication method. Note that *SUPI* also contains *idHN* and therefore identifies both a subscriber and its *HN*.

The 5G AKA Protocol [TS 33.501, Sec. 6.1.3.2] As mentioned before, the key K is used as a long-term shared secret, and *SQN* provides replay protection* for the subscriber. While *SQN* should be synchronized between the subscriber and the *HN*, it may happen that they become out-of-sync, *e.g.,* due to message loss. We thus use SQN_{UE} (respectively SQN_{HN}) to refer to the SQN value stored in the *UE* (respectively *HN*). The 5G-AKA protocol consists of two main phases: a *challenge-response* and an optional *re-synchronization procedure* (that updates the SQN on the *HN* side in case the *SQN* is out of-sync). A simplified protocol flow of 5G AKA is depicted in Figure 7.3.

Challenge-Response. Upon receiving a request for authentication material, the *HN* computes an *authentication challenge* built from:

- a random nonce R, *i.e.,* the challenge;
- a Message Authentication Code (MAC) M, proving the challenge's freshness and authenticity;
- *Res*, *i.e.,* expected *UE*'s response to the challenge that *HN* expects and *HRes*, *i.e.,* expected response that *SN* will expect from *UE*;
- K_{sess}, *i.e.,* session key for the secure channel that the subscriber and *SN* will eventually establish.

$\text{KDF}_1, \ldots, \text{KDF}_6$ are KDF functions completely unrelated with each other, and $\oplus$ denotes XOR.

M contains a MAC of the concatenation of R with the corresponding sequence number SQN_{HN} stored for this subscriber. A new sequence number is generated by incrementing the counter. The sequence number SQN_{HN} allows the subscriber to verify the freshness of the authentication request to defend against replay attacks and the MAC proves the challenge's authenticity. The *HN* does not send the challenge's full response

* This design choice is for historical reasons: old USIMs (*e.g.,* in 3G and 4G) did not have the capability to generate random nonces.

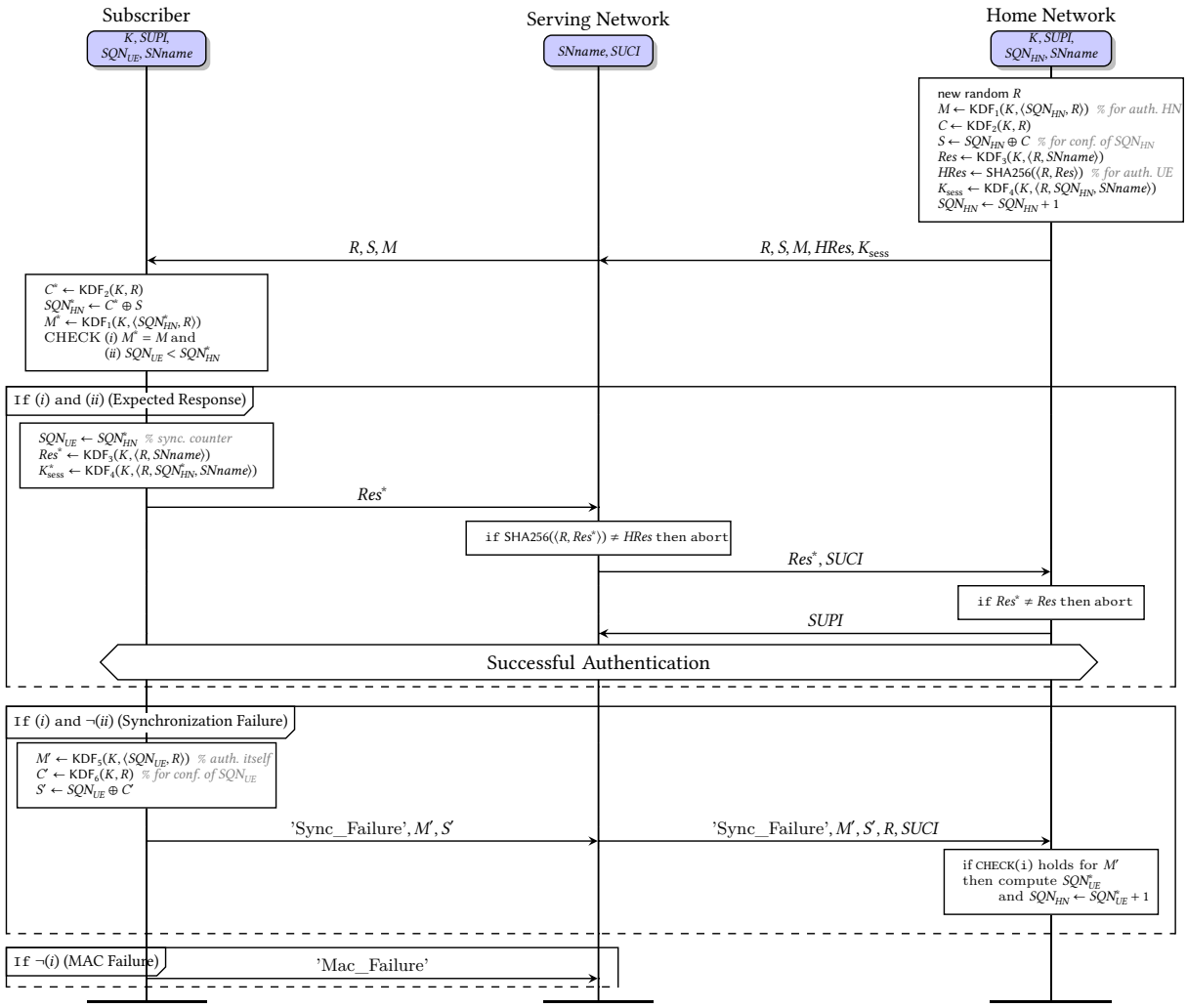


Figure 7.3.: The 5G AKA protocol, simplified presentation (continuing Figure 7.2)

*RES** to the *SN* but only a hash thereof; the rationale being that *HNs* are willing to have assurance of the presence of its subscribers even with malicious *SNs*.

The *SN* stores K_{sess} and the challenge's expected response and then forwards the challenge to the subscriber. Upon receiving the challenge, the subscriber first checks its authenticity and freshness. To do this, the subscriber extracts SQN^*_{HN} from *S* and checks that:

(i) *M* is a valid MAC value with respect to *K*, and replies 'Mac_failure' if it is not the case,
(ii) the authentication request is fresh[†], *i.e.*, $SQN_{UE} < SQN^*_{HN}$, and replies $\langle$'Sync_failure', M', $S'\rangle$ otherwise (these messages are explained in the re-synchronization procedure below).

If all checks hold, then the subscriber computes the key seed K_{sess}, which is used to secure subsequent messages. It also computes the authentication response *Res** and sends it to the *SN*. The *SN* checks that this response is as expected and forwards it to the *HN*, who validates it. If this validation succeeds then the *HN* confirms to the *SN* that the authentication is successful and sends the *SUPI* to the *SN*. Subsequent

[†] The freshness check may also consider non-normative protection against the wrapping around of SQN_{HN} which we do not describe here; see [TS 33.102, Sec. C].

communications between the *SN* and the subscriber can be secured using the key seed K_{sess}.

Re-synchronization procedure [TS 33.501, Sec. 6.1.3.2.1]. In case of a synchronization failure (case ¬(ii)), the subscriber replies with $\langle \text{'Sync_failure'}, M', S' \rangle$. The S' message enables the *HN* to re-synchronize with the subscriber by replacing its own SQN_{HN} by the sequence number of the subscriber SQN_{UE}; see [TS 33.102, Sec. 6.3.5,6.3.3]. However, SQN_{UE} is not transmitted in clear text to avoid being eavesdropped on (it is privacy sensitive as explained in Section 7.3). Therefore, the specification requires *SQN* to be *concealed*; namely, it is XORed with a value that should remain private: $C' = \text{KDF}_6(K, R)$. Formally, the concealed value is $S' = SQN_{UE} \oplus C'$, which allows the *HN* to extract SQN_{UE} by computing C'. Finally, the *UE* also sends M' allowing the *HN* to authenticate this message as coming from the intended subscriber.

Key-confirmation Note that the 5G specification considers some authentication properties to be *implicit*, in that authentication is supposed to be guaranteed only after an additional *key confirmation roundtrip* (with respect to K_{sess}) between the subscribers and the *SN*. We thus equip the model with an optional key-confirmation roundtrip where the subscribers and *SNs* confirm their key K_{sess} by MACing different constants. Our security analysis is then parametric in this roundtrip, allowing us to derive which properties hold without key confirmation, and what is gained by including this key confirmation step.

Other Variants of AKA Protocols EAP-AKA' is very similar to 5G AKA but has some differences in the message flow and in the key derivation functions (see [*Bas+18*]). The main differences between 5G AKA and earlier generation AKA protocols (3G and 4G) are in the Initiation of Authentication protocol (Figure 7.2), where the subscriber's identity in sent in clear text (allowing trivial passive tracking attacks).

7.2. Formal Analysis of 5G AKA [*Bas+18*]

We gave a precise and formal answer to the first research question. Namely, we apply formal methods and automated verification with Tamarin to determine precisely which security guarantees are met by 5G AKA.

Unfortunately, the AKA protocols, and *a fortiori* 5G AKA, feature a combination of properties that were extremely challenging for state-of-the-art verification techniques and tools at the time and, until some of our enhancements we proposed at the time, a detailed formalization was outside of their scope. First, the flow and the state-machines of these protocols are large and complex. This is due in part to the use of sequence numbers (*SQN*) and the need for a re-synchronization mechanism should counters become out-of-sync. This complexity is problematic for tools that reason about a bounded number of sessions as they scale poorly here. It also eliminates the option of machine-checked manual proofs as the number of interactions is too large for humans to explore. Second, these protocols are stateful (the *SQN* counters are mutable and persist over multiple sessions) and have numerous loops. This makes inductive reasoning necessary and rules out fully automated tools, which are not yet capable of automatically finding appropriate inductive invariants. Finally, the AKA protocols use the Exclusive-OR (XOR) primitive to conceal some

values. This primitive is notoriously hard to reason about symbolically, due to its algebraic properties (*i.e.,* associativity, commutativity, cancellation, and neutral element). For this reason, prior works provided only limited models of the AKA protocols, which were insufficiently precise for a satisfactory analysis, see [*Bas+18*] for details. Given these features, we were left at the time with just the verifier Tamarin [Mei+13] as a suitable tool, and we had just extended Tamarin to handle XOR [*Dre+18*] (see Section 4.1.1).

Yet, Tamarin fails to prove relevant security properties and even extremely weak properties such as the full executability of the protocols. Our work straddles this divide: we developed a proof structure based on intermediate lemmas (called *helping lemmas*) as well as proof strategies for proving these lemmas and the security properties. Proof strategies are implemented through *oracles* that offer a light-weight tactic language, implemented in Python, to guide the proof search in Tamarin.

We focus here on our analysis of the authentication, agreement and confidentiality properties of 5G AKA. Although we also have analyzed some privacy properties, we omit them here because they were limited in scope at the time and we explain why it is still challenging to do so today in Section 7.4.3.

7.2.1. Summary of our 5G AKA Analysis

We summarize next our three main contributions in this work: our formalization, models, and analysis results.

Formalization of the 5G Standard We extract and formally interpret the standard's security assumptions and goals. In doing so, we identify key missing security goals and flaws in the stated goals, in particular the standard does not aim to achieve agreement on the session key and on the *SN*. We target a wide range of properties — confidentiality, authentication, and privacy — and their fine-grained variants. This required considerable analysis and interpretation of the 3GPP Technical Specification (722 pages across 4 documents). We do not detail this contribution here and refer the reader to [*Bas+18*] for details.

Formal Model of 5G AKA We have also tackled the aforementioned challenges to provide the first faithful model of an AKA protocol that is detailed enough for a precise security analysis and is still amenable to automation. The modeling choices for formalizing our interpretation of the standard were crucial. To support reasoning about our model, we developed dedicated proof techniques based on inductive lemmas and proof strategies that guide proof search. We refer the reader to [*Bas+18*] for details.

Security Evaluation of 5G AKA We have then carried out the first formal security evaluation of 5G authentication at the time, providing a comprehensive analysis of the 5G AKA protocol. This includes:

- *a formal, systematic security evaluation*: we leverage our model of 5G AKA to automatically identify the minimal security assumptions required for each security goal to hold. We found that some critical authentication properties were violated prior to key confirmation,

which is not clearly mandated by the standard. Some other properties are not met, except under assumptions on the 5G ecosystem that are missing from the standard.

- *recommendations*: we make explicit recommendations and propose provably secure fixes for the attacks and weaknesses we identified. Most of our recommendations generalize to 5G Authentication as a whole, and not just 5G AKA.

We summarize our analysis key results in Section 7.2.2 and summarize some of our recommendations in Section 7.4.1 and refer the reader to [*Bas+18*] for more details.

7.2.2. Selection of Key Results of our 5G AKA Analysis

We present some key results from our comprehensive analysis of the 5G AKA protocol. We emphasize that we *automatically* analyze the *formal security guarantees* that the protocol provides for an *unbounded number* of sessions executed by honest and compromised subscribers, *SNs*, and *HNs* when used *in combination* with the initiation protocol. Thus, our analysis accounts for all potential unintended interactions an attacker could exploit between these sub-protocols run by all possible instantiations of the three roles we consider.

We depict the outcome of our analysis of authentication properties in Table 7.1. For each pair of parties, we present the minimal assumptions required to achieve authentication properties: *i.e.,* weak agreement, non-injective agreement, and injective agreement. We only consider agreement on relevant data; *i.e.,* K_{sess}, *SNname*, and the *SUPI* (recall that the *SUPI* already contains id_{HN}). The assumptions are minimal in that strengthening the attacker's capabilities in any direction violates the property. The symbol ✗ denotes that the property is violated for the weakest threat model where all participants are honest, none of the compromise scenarios is considered, and key confirmation is systematically enforced. Similarly, we present our results concerning secrecy properties in Table 7.2. We depict results with [·] when they correspond to a security property explicitly listed as a security goal in the standard.

We only check for 2-party authentication properties, which expresses well the security goals of 5G AKA. Note however that we obtain a form of 3-party agreement property (where all 3 parties' views coincide) as a corollary of three 2-party agreement properties. This is because we check for strong 2-party agreement properties on several data points and identifiers simultaneously.

Discussion Table 7.1 clearly shows the extent that the 5G standard underspecifies authentication requirements; *i.e.,* some properties with [·] rely on extra assumption. We also indicate a number of properties that are violated even in the best-case scenario (✗). We discuss why in Section 7.2.2.1. Afterwards, we explain and critique the use of key confirmation in Section 7.2.2.2. We discuss privacy properties in Section 7.4.2. Finally, our results concerning secrecy properties are as expected and are not discussed further. Also, PFS of K_{sess} is violated as expected.

7.2.2.1. Missing Security Assumption

The 5G AKA protocol fails to meet several security goals that are explicitly required as well as other critical security properties. This is still true under

Table 7.1.: Minimal assumptions required for 5G AKA to achieve authentication properties. We denote subscribers by *UE*, non-injective by NI, and injective by I. Assumptions are expressed in terms of forbidden reveals (*e.g.*, $\neg K$, meaning the property only holds when K is not revealed). We also indicate whether a key confirmation phase is needed with k-c while $\neg$ch denotes an uncompromised channel between *SN* and *HN*. When not otherwise specified, the worst-case scenario is considered; that is K,*SQN*,*SUPI*,sk_{HN}, and the channel between *SN* and *HN* are compromised and the key confirmation phase is skipped. ×: the property is violated *by definition* (*e.g.*, because *SUPI* is constant). wa: the property coincides with weak agreement and requires same assumptions. The explicit goals given in the specification are denoted by [·] around them.

Point of view	*UE*				*SN*				*HN*			
Partner	*SN*		*HN*		*UE*		*HN*		*UE*		*SN*	
Agreement	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I
on K_{sess}	✗	✗	$\neg K \wedge$k-c	$\neg K \wedge$k-c	✗	✗	$\neg$ch	$\neg K \wedge \neg$ch	$\neg K$	$\neg K$	$\neg$ch	$\neg$ch
on *SUPI*	wa	×	wa	×	wa	×	[$\neg$ch]	×	wa	×	×	×
on *SNname*	wa	×	[$\neg K \wedge$k-c]	×	wa	×	wa	×	[$\neg K$]	×	wa	×
Weak agreement	[✗]		$\neg K$		[$\neg K \wedge \neg$ch]		$\neg$ch		$\neg K$		$\neg$ch	

Point of view	*UE*	*SN*	*HN*
K_{sess}	$\neg K \wedge \neg$ch	$\neg K \wedge \neg$ch	$\neg K \wedge \neg$ch
$PFS(K_{\text{sess}})$	✗	✗	✗
SUPI	$\neg sk_{\text{HN}} \wedge \neg$ch*	–	$\neg sk_{\text{HN}} \wedge \neg$ch*
K	∅	∅	∅

Table 7.2.: Minimal assumptions for 5G AKA to achieve secrecy properties. We omit the assumption that data that is supposed to be secret is not revealed. See Table 7.1 for the legend. The symbol * indicates that there is no dishonest *SNs* at all and the underlying property is always violated otherwise. PFS(·): perfect-forward secrecy. –: property not relevant.

the assumptions specified in the standard, even after a successful key-confirmation phase (see ✗ in Table 7.1).

More specifically, the agreement properties on K_{sess} between the subscribers and *SNs* are violated. So is weak agreement from the subscribers towards the *SNs*. This is caused by the lack of a binding assumption on the channel between *SNs* and *HNs* and because the *SUPI* is sent to the *SN* in a different message than the message containing K_{sess}, which is sent earlier. Therefore, as soon as a pair of an *SN* and an *HN* runs two sessions concurrently, there is no assurance that the *SUPI* the *SN* receives at the end of the protocol actually corresponds to the K_{sess} it has received earlier (it could correspond to another concurrent session). As a consequence, an *SN* may associate the session key K_{sess} to the correct subscriber (a necessary condition for the key confirmation to be successful), but to the wrong *SUPI*, violating the aforementioned properties. In practice, this could allow an attacker to make the *HN* bill someone else (*i.e.,* with a different *SUPI*) for services they consume from an *SN* (*i.e.,* encrypted with K_{sess}). Thus, the binding property for the channel between the *SNs* and *HNs* appears to be a critical security assumption, and should be explicitly mentioned in the standard. This weakness has been introduced in the version v0.8.0 of the standard (published in March 2018). In the previous version (v0.7.1), the *SUPI* was sent by the *HN* to the *SN* together with the challenge[1], thus preventing the aforementioned attack and making the binding assumption unnecessary. However, the final version of the standard requires this additional assumption. A similar looking issue, but between two parts of the *HN* has been previously observed by [CD19], but it is an entirely different concern than the one we describe.

Table 7.3 depicts additional security properties the 5G AKA protocol provides when the channel between the *SNs* and *HNs* is assumed to be binding. Under this assumption, the previously violated properties are now satisfied under reasonable threat models. We only show results for *UEs* and *SNs* to show how their guarantees change.

1: To the best of our knowledge, the rationale behind the new version is to let *HN* wait for the proof of the subscriber's recent aliveness before disclosing the corresponding *SUPI* to *SNs* that may be malicious or dishonest.

P.o.V.	*UE*		*SN*	
Partner	*SN*		*UE*	
Agre.	NI	I	NI	I
on K_{sess}	$\neg K \wedge$k-c$\wedge\neg$ch	$\neg K \wedge$k-c$\wedge\neg$ch	$\neg K \wedge \neg$ch	$\neg K \wedge \neg$ch
Weak agre.	[$\neg K \wedge$k-c$\wedge\neg$ch]		[$\neg K \wedge \neg$ch]	

Table 7.3.: Minimal assumptions required for 5G AKA to achieve authentication properties between *UEs* and *SNs*, assuming that the channel between *HNs* and *SNs* is binding. Agreements on *SUPI* and *SNname* are not impacted.

7.2.2.2. Implicit Authentication

A successful key-confirmation roundtrip is required to obtain crucial security guarantees. More precisely, this roundtrip is required for all agreement properties from the subscribers' point of view except weak agreement towards the *HNs*. Indeed, an attacker can impersonate an *SN* towards a subscriber but is unable to learn the K_{sess} key the subscriber has computed.

> [TS 33.501, **Sec. 5.1.2**] The meaning of 'implicit key authentication' here is that authentication is provided through the successful use of keys resulting from authentication and key agreement in subsequent procedures.

The 5G standard only requires *implicit* authentication properties for the subscribers. However, the standard neither specifies that subscribers must wait for this key confirmation to be successful before continuing nor does it specify this additional roundtrip *as part* of the authentication method. As a consequence, the standard makes a choice that we consider risky: it postpones the handling and the verification of the additional key confirmation roundtrip to *all possible subsequent procedures* (*e.g.,* the NAS security mode command procedure [TS 33.501, Sec. 6.7.2]). The standard fails to specify a standalone authentication protocol that provides a reasonable set of security guarantees since some critical properties are provided only when the protocol is used in specific, appropriate contexts.

More importantly, since the standard makes the overall security of the authentication rest on subsequent procedures, it is very challenging, and out of the scope of our analysis, to assess if all currently specified subsequent procedures (as well as future ones that may be added) either correctly mandate the use of this key confirmation roundtrip or do not require authentication properties from the subscribers' point of view towards the *SNs*. We believe that there are at least two potential use cases where the above weakness could represent a vulnerability. First, the standard specifies that *SNs* can initiate *key change on-the-fly* [TS 33.501, Sec. 6.9.4.1] as well as switch security contexts [TS 33.501, Sec. 6.8], including keys, parameters, *etc.* This raises the question whether a malicious *SN* or a fake base station could not *fully* impersonate a genuine *SN* towards the subscribers by changing the session key immediately after 5G AKA. Second, in a scenario where subscribers use the presence of *SNs* for geolocalization or for making sensitive decisions (related to *e.g.,* emergency calls), an active attacker could impersonate an *SN* since the (mismatched) K_{sess} key may not be needed or used.

Finally, the key confirmation roundtrip is not the only option to achieve the aforementioned missing security guarantees. We provide and discuss in Section 7.4.1 two alternative solutions that fix this issue while reducing necessary communications.

7.3. New Privacy Risks and Practical Attacks [*OBH17*; *Bor+19*]

In other, earlier works, we addressed the second and third research questions through manual and formal analysis of privacy properties in AKA protocols [*OBH17*; *Bor+19*].

7.3.1. Privacy Risks over WiFi

First of all, we analyzed the EAP-AKA protocol and most common configurations used by smartphones to connect to mobile services through WiFi [*OBH17*]. We have noticed widespread use of specific authentication protocols, in particular EAP-AKA, that have failed to employ identity privacy features and transport the IMSI unencrypted over WiFi. Crucially, we have discovered that these protocols are preconfigured on many smartphones so that devices automatically attempt authentication. During the course of the protocol exchange the IMSI may be observed or, with an active attack, can be forcibly revealed.

The use of these protocols had seen rapid growth at the time as they enable automatic authenticated WiFi connection in public spaces (such as the London Underground). Furthermore we have also developed another technique that performs an active attack that enables the extraction of the IMSI from phones that have enabled the "WiFi-Calling" feature which was being offered by a growing number of operators. In this context, the leakage of the IMSI allows linking it with the user's other hardware addresses, such as the WiFi MAC address, so tracking may then be performed independently of IMSI extraction attacks.

We have implemented both the active and passive attacks which may be deployed on a laptop, or embedded device, with a suitable WiFi interface. These privacy threats have arisen from a number of competing issues that arise from the trade-offs and compromises made in the protocol designs and in the actual deployment of these protocols into the mobile networks. We had prolonged discussions with both the mobile OS manufactures and mobile operators in an attempt to address the issues before we went public. In [*OBH17*] we detail our experiences, examine the issues and attempt to address them. After following the responsible disclosure process, our research findings were acknowledged by the GSMA and device manufacturers (Google, Apple, Microsoft, Blackberry). As a direct result of our reporting these issues to Apple they decided to develop and deploy the privacy enhanced *conservative peer* mode for EAP-SIM/AKA into iOS10. We redirect the curious reader to [*OBH17*] for more details.

7.3.2. Privacy Risks of AKA in 3G, 4G, and 5G

Second, we manually analyzed the AKA protocols used in 3G, 4G, and upcoming 5G networks [*Bor+19*]. We have revealed a new privacy attack against all variants of the AKA protocol (including 5G AKA and EAP variants) that breaches subscribers' privacy more severely than known *subscriber location privacy attacks* do. Our attack exploits a new logical vulnerability we uncovered in the protocol specification that requires dedicated fixes. We briefly summarize and discuss these findings in the following sections.

7.3.2.1. Summary of our Privacy Analysis and Attacks

New Logical Attack. We found a new logical vulnerability in the specifications of all aforementioned variants of AKA: the protection mechanism of the SQN can be defeated under specific replay attacks due to its use of XOR and a lack of randomness. We show how to leverage this vulnerability to break the confidentiality of SQN, thus defeating the purpose of a dedicated protection mechanism and breaking an explicit privacy requirement [3GPb]. We show that partly learning SQN leads to a new class of privacy attacks (*i.e., activity monitoring attacks*): an active attacker can leverage fake base stations and our attack to learn information about targeted subscribers' mobile service consumption, even when subscribers move away from the attack area (*e.g.,* range of a fake base station). This is in stark contrast to location attacks that do not reveal service consumption and requires the targeted subscribers to stay in attack areas. Less importantly, we show that our logical vulnerability also yields a new location attack. We briefly summarize the key aspects of this logical attack and its practical impact in Section 7.3.2.2.

Low-cost proof of concept. We demonstrate the feasibility of our attack using widely available and low-cost setup on commercial 4G networks in several European countries. Our attack affects all 3G and 4G devices currently deployed all over the world and future 5G devices (according to the specification [3GPc]). We shortly describe our experimental setup and practical impact in Section 7.3.2.3.

Security analysis and countermeasures. We discuss the weaknesses of the AKA protocol, its deployment, design trade-offs, and the overall cellular architecture that have made possible our attack in order to draw lessons for the future generation networks. We propose countermeasures to our attack and establish formal security guarantees using the state-of-the-art tool Tamarin [Mei+13]. As we shall see in Section 7.4.4, 3GPP and GSMA have acknowledged our findings and have eventually included a fix in the 5G standard. We refer the reader to [*Bor+19*] for more details and only summarize some key lessons and recommendations in Section 7.4.2.

7.3.2.2. Logical Vulnerability in SQN Protection

Our attack vector exploits a lack of randomness and the use of XOR in the concealed sequence number $S' = SQN_{UE} \oplus C'$ where $C' = \mathsf{KDF}_6(R, K)$. The value R is part of the challenge R, S, M sent by *HN* to *UE* and is supposed to be randomly chosen. However, if the *UE* receives two times the same challenge R, S, M with the same R and yield two synchronization failures, then the two concealed *SQNs* will be of the form: $S'_1 = SQN^1_{UE} \oplus C'_1$ and $S'_2 = SQN^2_{UE} \oplus C'_2$ such that

$$C'_1 = \mathsf{KDF}_6(R,\ K) = C'_2.$$

Therefore, an attacker having a genuine challenge R, S, M for some *UE* can transmit it to the *UE* at two different times t_1 and t_2, retrieve values S'_1 and S'_2, and compute:

$$\begin{aligned} S'_1 \oplus S'_2 &= (SQN^1_{UE} \oplus C'_1) \oplus \\ &\quad\ (SQN^2_{UE} \oplus C'_2) \\ &= SQN^1_{UE} \oplus SQN^2_{UE} \end{aligned}$$

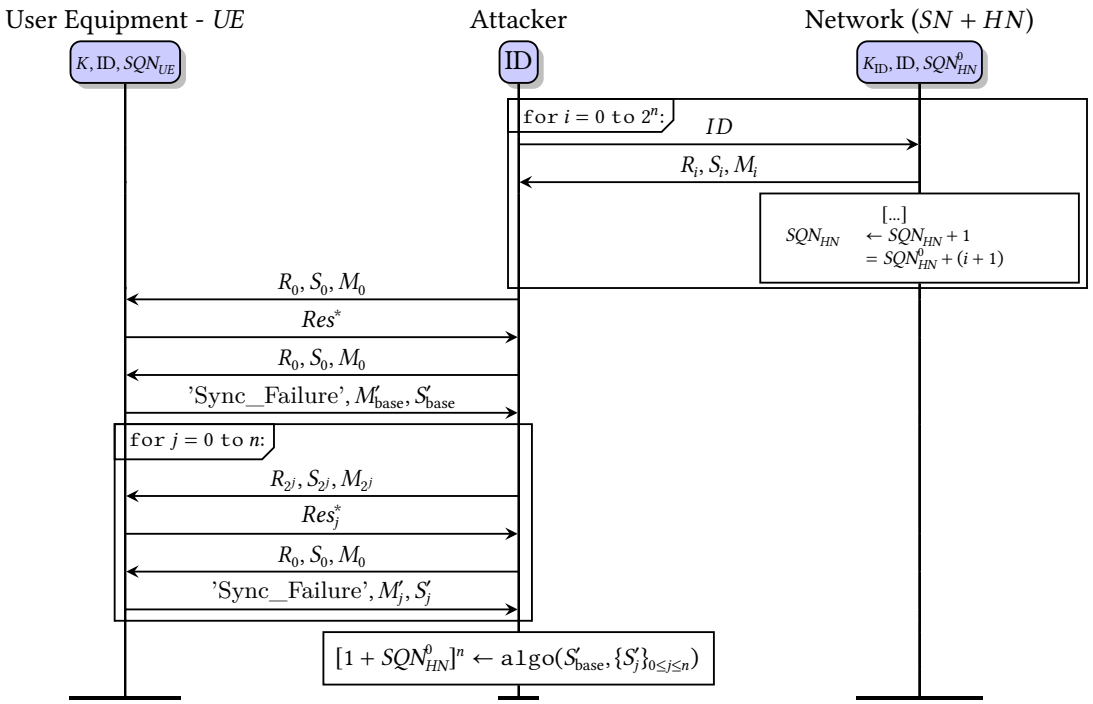


Figure 7.4.: Sequence Number Inference Attack. ID is a subscriber identifier, *IMSI* for 3G and 4G or an encryption thereof in 5G. SQN^0_{HN} is the initial *SQN* for ID stored in the *HN* and $[X]^n$ denotes the n least significant bits of X.

where SQN^i_{UE} is the value SQN_{UE} at time t_i. We shall see that by cleverly choosing several timestamps t_i's, the attacker is able to exploit values such as $SQN^i_{UE} \oplus SQN^j_{UE}$ to break the confidentiality of SQN.

Breaking the Confidentiality of SQN We show how an active attacker who knows any *UE*'s identity (temporary, permanent, or encrypted as in 5G with *SUCI*) is then able to learn the n least significant bits of SQN_{HN}, stored in the *HN*. We depict the main steps of our attack in Figure 7.4. Note that n does not necessarily correspond to the full bit-length of *SQN*, *i.e.,* 43 to 48 bits depending on the operator, since practical privacy attacks can be mounted with only a few bits of *SQN* leakage (see Section 7.3.2.3).

First phase: fetching challenges. In a first phase (loop for $i = 0$ to 2^n from Figure 7.4), the attacker needs to fetch consecutive challenges R_i, S_i, M_i intended for the targeted *UE*. This is made possible by the fact that, in the AKA protocol, *UE* receives such challenges prior to authentication but after identification. Therefore, an attacker only needs to know one valid identity of the targeted *UE* (*e.g., IMSI* or encrypted permanent identities such as *SUCI*) in order to get these challenges. Note that because SQN_{HN} is incremented by 1 after the computation of every challenge, S_i is computed based on some SQN value (that we denote by $SQN_{HN}(S_i)$) equals to $SQN^0_{HN} + i$. We later describe a variant of this attack where this phase can be performed fully passively. We will explain in Section 7.3.2.3 how this can be easily done in practice.

Second phase: challenge injections. Immediately after the first phase, the attacker injects the first challenge they obtained: R_0, S_0, M_0. From the *UE*'s perspective, this is a genuine challenge (the MAC verification (i) succeeds) that has never been received before and that is based on a recent enough $SQN_{HN}(S_0 0) = SQN^0_{HN}$ (the freshness verification (ii) succeeds). At this time (before the second loop), SQN_{UE} equals $SQN^0_{HN}+1$. Then, the attacker

injects again the challenge R_0, S_0, M_0 yielding a synchronization failure containing some $M'_{\text{base}}, S'_{\text{base}}$ message where the conceal SQN equals:

$$S'_{\text{base}} = (SQN^0_{HN} + 1) \oplus \mathsf{KDF}_6(R_0, K).$$

Finally (loop for $j = 0$ to n from Figure 7.4), the attacker injects $R_{2^j}, S_{2^j}, M_{2^j}$ that is accepted by the *UE*, in order to make the *UE* updates its SQN_{UE} to the value

$$SQN_{UE} := SQN_{HN}(S2^j) + 1 = SQN^0_{HN} + 2^j + 1.$$

After each such injection, the attacker then injects again the challenge R_0, S_0, M_0 provoking a synchronization failure containing some M'_j, S'_j where:

$$S'_j = (SQN^0_{HN} + 2^j + 1) \oplus \mathsf{KDF}_6(R_0, K).$$

Third phase: off-line inference algorithm. We now describe $\mathtt{algo}(\cdot)$ that takes the $n + 2$ fetched S' messages (*i.e.,* S'_{base}, S'_j for $0 \leq j \leq n$) as inputs and outputs the n least significant bits of $1 + SQN^0_{HN}$. Recall that $S'_{\text{base}} = (1 + SQN^0_{HN}) \oplus \mathsf{KDF}_6(R_0, K_{IMSI})$ and $S'_j = (1 + SQN^0_{HN} + 2^j) \oplus \mathsf{KDF}_6(R_0, K_{IMSI})$. Therefore, for any $0 \leq j \leq n$, it holds that:

$$S'_{\text{base}} \oplus S'_j = \quad (1 + SQN^0_{HN}) \oplus (2^j + 1 + SQN^0_{HN}).$$

We note δ_i this quantity $S'_{\text{base}} \oplus S'_j$. One has that $\delta_i = (2^i + X) \oplus X$ for all $0 \leq i \leq n$ where $X = 1 + SQN^0_{HN}$ is the quantity we seek to infer the n least significant bits of. In a nutshell, the idea of the algorithm consists in analyzing how remainders propagate in $(2^i + X)$ at bit position i and $i + 1$ (in little-endian notation) by looking at δ_i; see [*Bor+19*] for details. Note that this algorithm can be executed completely offline on the collected data.

Improving the Attack Under Stronger Threat Model Note that under certain circumstances (*i.e.,* when the *UE* is performing a lot of authentication sessions when in the attack area), a far less costly variant of the attack is possible with only $n + 2$ active interactions with the *UE*, achieving the same goal. The attacker can simply passively observe the *UE* running 2^n authentication sessions with the *HN* while collecting the authentication tokens, hence avoiding the first phase of the attack.

7.3.2.3. Practical Privacy Attacks and Impact

We explained how a logical vulnerability in AKA can be exploited to break the confidentiality of the *SQN* counter. We now explain how one can leverage this attack to break subscribers' privacy by conducting an *activity monitoring attack*. This new class of attacks allows an attacker to monitor targeted subscribers' activity, even for periods where subscribers *escape the attack area.*

In a nutshell, the attacker needs to conduct the previously described attack when targeted subscribers are in the attack area, thereby learning n significant bits of *SQN* at different times $t_1, t_2, \ldots$. Because *SQN* is incremented after each successful authentication, the differences of *SQN* values obtained at times $t_1, t_2, \ldots$ reveal the number of AKA sessions the targeted subscribers have performed between these times. We have explored in [*Bor+19*] practical considerations and measurements in the wild

showing that the attacker can then relate the number of AKA sessions some *UE* has performed in a given period of time to its typical service consumption during that period. We also show in [*Bor+19*] a variant of our attack that could yield location attacks in a variant of AKA, even when fixed with countermeasures for all known privacy attacks.

Examples of Practical Scenarios We now illustrate the potential real-life impacts of our activity monitoring attack with two practical scenarios.

Spying on embassy officials, journalists, or any high-value target. Assuming an adversary having a fake base station near an embassy, they can learn the officials' activity not only when they are at the office during working hours, but also when they are not, including during evenings and nights (*e.g.,* at home) or during business trips. Therefore, such an attacker may learn if targets use different SIMs cards for private use (no activity at home), if some specific time periods (*e.g.,* one evening and night) were specifically busy (a lot of calls or SMSs were made yielding a big rise of SQN), if one is using their work phone at home, if a phone was switched off for certain time periods or trips (possibly indicating multiple SIMs usage in case very little or no activity was detected for a long period out of town[2]), *etc.*

Better ads targeting. Consider for instance a shop that is willing to know more about its customers (*e.g.,* for improving ads targeting) using fake base stations. This kind of scenario has already been reported [ME12] (using Wi-Fi capabilities of smartphones) and exploited [CH13] in real shops. Our attack causes a new threat in that context since it leaks to the shop typical customers' mobile consumption during time periods *between* customers' visit (while they escape the attack area).

2: Note that such scenarios are even more likely today with the large availability of ESIM cards, which greatly simplify SIM cards switching.

7.3.2.4. Proof of Concept

We now show how we have performed our attacks in practice on 4G networks using a low-cost and easily available setup. We then explain practical aspects which make our attack easily feasible (*e.g.,* issues in different operator's network and security configurations). We finally discuss our Proof-of-Concept (PoC) and our experimental results.

Remark 7.3.1 (On 5G.) Due to unavailability of 5G devices and networks at the time, we have only focused on 4G environments. We stress however that, as explained later, our attacks were also valid for 5G, before the dedicated fix was implemented (see Section 7.4.4). Indeed, the only major difference in 5G concerns the way we would fetch authentication challenges. To do so, we do not have to learn *SUPI* (the subscribers' identifier in 5G that is supposed to remain secret) but only a valid *SUCI*. Since *SUCI* is sent in the clear by subscribers, the attacker just has to eavesdrop on one *SUCI* and can from then on, fetch as many authentication tokens for that subscriber as desired (using *SUCI* instead of *IMSI*). The rest of the attack remains the same for 5G infrastructure. ▷

Experimental Setup The first experimental setup aims at building a platform to collect the victim's authentication challenges. Further, we modify and build software tools to make victim's UE to attach to a rogue 4G base station so that one can inject legitimate radio layer signaling messages. Our hardware setup is depicted in Figure 7.5 which consists of a laptop, a Universal Software Radio Peripheral (USRP) B210 [Ett], and a PC/SC [PCS] capable smartcard reader (we used ACS ACR38 [Adv]) with commercial USIM cards. At the time we conducted our experiments (in 2018), our complete experimental setup costs about 1140€ excluding a laptop price (that could be replaced by a cheaper Raspberry pi) - 1120€ for USRP, 20€ for commercial operator's prepaid USIM cards.

Attacker's Setups We now explain how hardware and software components can be combined into different setups to demonstrate our attack.[3]

Obtaining authentication challenges. We used the software *srsUE* from the *srsLTE* suite [Gom+16] configured with the target's IMSI with the

3: We discuss ethical aspects of our experiments in Section 7.4.4.

USRP B210 for obtaining authentication challenges. Essentially, the USRP B210 tries to impersonate the target's USIM. When doing so, each session fails because *srsUE* does not know the target's secret key K (so it cannot compute the appropriate Res^*) but, before the failure, we obtain a new, genuine authentication challenge that is intended for the target's USIM. We were able to fetch authentication tokens using the USRP at a surprisingly high speed (see discussion later) but, if for some reason, a network recognizes the USRP as a fake smartphone, we can still use genuine phones with programmable [Sys] USIM cards (ca. 80€). We have also tested a simpler but less flexible setup using only a smartphone and reprogrammable USIM card without USRP [*Bor+19*].

Fetching re-synchronization messages using rogue base station. Utilizing OpenLTE [Woj] based 4G network running on an USRP B210, we sniff over-the-air signaling messages and masquerade as a real base station. We configure a rogue base station to mimic a real operator to lure victim's UE to attach by 4G reselection procedure [Sha+16]. The base station is then able to inject messages and eavesdrop on replied messages. We use this method to fetch the S'_j re-synchronization messages that a USIM sends as part of the AKA protocol.



Figure 7.5.: Our experimental setup, showing a smartcard reader, USRP (left), set of commercial USIM cards, and a test phone.

Attack Background Before explaining our PoC, we report on our investigation on AKA related security configurations of 4G networks which make our attack easier to perform. We selected several major European 4G operators including three German, three Austrian, two French, and one Swiss operators.

We were successfully able to collect authentication challenges intended for the targeted USIM at any moment for any subscriber in the world. Note that to achieve this step, the attacker only needs to know the IMSI (or any temporary or encrypted identity) of that particular victim's USIM. If the attacker knows the subscriber's mobile phone number, they can perform HLR Lookup attacks [Eng] to learn victim's IMSI. The previous work [Sha+16] also demonstrates how to find IMSI and temporary identifier (GUTI) of the targeted victim by knowing the mobile phone number or social identities such as email, Facebook and Twitter. Based on the data collected from our experiments, we studied the following parameters of the operator's 4G networks. We stress that there is no need to learn more information (*e.g.*, private key K_{IMSI}) about the targeted USIM.

We found that most operators allowed authentication challenges to be fetched without a rate limit. Using our first setup using *srsUE*, we were able to fetch fresh, unused authentication challenges consecutively at the speed of 1 per second. Using our second setup involving a smartphone, we were able to fetch more than 30 challenges in less than 10 minutes. We expect a setup based on multiple rogue base stations to achieve much better performance.

Full Proof of Concept Upon knowing the victim's identity and location [Sha+16], an attacker can perform our subscriber activity monitoring attack. This attack requires obtaining a larger number of authentication challenges of victim's USIM. But as discussed earlier, we did not observe any countermeasure preventing us to fetch a large amount of them. Exploiting our attack, the more consecutive challenges one fetches, the more bits the attacker can infer from the *SQN* of the victim. Then, using a rogue base station, the attacker is then able to inject parts of those challenges and store replied S^* as explained in Section 7.3.2.4.

We build a modest setup as a proof of concept of our attack. We were able to request 1025 authentication challenges and collected 12 concealed *SQN* values S' from 24 injections of AKA messages. Using our generic *SQN* inference algorithm, those 12 S' messages were enough to infer at least 10 bits of *SQN* (the least significant ones), sometimes more[4]. Obviously, an attacker with greater capabilities and more elaborate setups (notably multiple rogue base stations for fetching challenges; which turns out to be the bottleneck) could infer more bits.

4: Our generic algorithm from [*Bor+19*] can be more efficient than the one described earlier.

Impacted devices. The AKA protocol vulnerability we found is part of the 3GPP specifications and does not rely on implementation issues in 4G/3G devices. In fact, the affected AKA protocol is implemented in the USIM and not in the baseband OS of devices. Thus, any 3G/4G device deployed worldwide having active USIM card is affected by our attacks. For our investigations, we selected prepaid USIM cards of few leading cellular operators. We collected and stored unused authentication challenges of related USIM cards as described before. Then we successfully verified that these USIM cards were vulnerable to our attack. As mentioned earlier about feasibility in 5G networks, if no dedicated mechanism for mitigating our attack is implemented, 5G devices will also suffer from our attack.

Range of the attack. Previous research suggested that the coverage radius of a rogue base station using USRP B210 and OpenLTE ranges between 50 and 100 meters [Sha+16], without external hardware to boost the signal. The coverage range of our attacks could be increased to locate and inject radio layer messages to a 4G device equipped with USIM within a 2 km^2 area as discussed in [Sha+16].

Detection. We also investigated possible methods for the end subscribers to detect our attack. Unfortunately, our attacks cannot be detected by the mobile OS (*e.g.,* Android or iOS). The reason is that the AKA protocol is executed in USIM and the baseband chip that communicates limited information to the mobile OS.

7.4. Lessons Learned and Discussions with GSMA and 3GPP

7.4.1. Recommendations for the 5G AKA Standard

We now summarize the weaknesses in the standard and suggestions for improvements and refinements. We also propose provably secure fixes as a replacement for the key confirmation and the binding channel assumptions. Again, we emphasize the critical role played here by our formal interpretation of the standard and our formal analysis of the described 5G AKA protocol.

Explicit Requirements The standard underspecifies security requirements for the 5G AKA protocol. We suggest that the standard explicitly requires the missing intended security properties. In particular, it should be clear that 5G AKA aims at achieving **injective agreement on K_{sess} between the subscribers and the *SNs*** which is central to the protocol's purpose. The subscribers should obtain **injective agreement on K_{sess} with the *HNs***; they are thereby assured the *HNs recently* authorized this session, since K_{sess} is derived from the random R. Finally, the *HNs* should have **injective agreement on K_{sess} with the subscribers**, obtaining recent aliveness as a consequence. We also detail in [*Bas+18*] missing cryptographic

assumptions for the functions $\mathsf{KDF}_1, \ldots, KDF_6$ (which are not explicitly stated in the standard to be KDF functions) and some lack confidentiality properties.

Binding Channel As discussed in Section 7.2.2.1, an update in the standard introduced attacks under the given security assumptions. There are two solutions to fix this: either the standard explicitly states an additional security assumption (*i.e.,* the channel between the *SNs* and *HNs* must be binding), or alternatively the 5G AKA protocol is fixed (without the need for a new assumption) using the following minor modification: $\langle SUPI, SUCI \rangle$ is sent instead of *SUPI* in the final message from *HN* to *SN*. However, it is our understanding that the binding assumption is a property that is required for other reasons anyway, such as reliability.

On Key Confirmation We already have discussed the danger of missing key confirmation in 5G AKA in Section 7.2.2.2. We now propose two simple modifications to the protocol that would make key confirmation redundant and unnecessary, therefore reducing the number of roundtrips that are needed to achieve intended security guarantees. Before explaining our fix, note that the key confirmation was necessary in the first place because the *HNs* never commit to a specific *SNname* when computing the challenge R, S, M. Only the key K_{sess} is bound to *SNname*, but the challenge itself is not.

Our first fix consists of binding M to *SNname* so that subscribers directly have the proof the *HN* has committed to a specific *SNname*, without even using K_{sess}. Formally, M currently refers to $\mathsf{KDF}_1(K, \langle SQN_{HN}, R \rangle)$. In our fix, *MAC* is replaced by $\mathsf{KDF}_1(K, \langle SQN_{HN}, R, SNname \rangle)$. Therefore, the subscribers can verify the authenticity of the challenge that commits to a specific *SNname*. We have formally verified that a key-confirmation roundtrip is no longer necessary with this fix.

Our second, alternative, fix consists in replacing the full key-confirmation roundtrip by an unidirectional key confirmation from the *SN* only. More precisely, we could add (any) message MACed with a key derived from K_{sess}, sent by the *SNs* to the subscribers, at the very end of the protocol. We have proven with Tamarin that no further guarantees are provided by a full key confirmation, compared to our (less costly) unidirectional key confirmation.

Redundancies A close look at the cryptographic messages and their purposes shows many redundancies. For instance, in RES^*, the proof of possession of K is in *CK*, *IK*, and *RES*. R appears to be redundant as well. Similarly, *SNname* is redundant in the key derivation of K_{sess}. Legacy reasons may explain these redundancies, but these design choices could be questioned and the protocol simplified.

7.4.2. Recommendations Regarding Privacy

We recommend for the standard to explicitly aim at protecting privacy against active attackers and take steps in this direction.

Fixing our *SQN* Confidentiality Attack As already explained, the main attack vector we exploit in our confidentiality attack is the use of XOR and the lack of subscriber-generated randomness in C' making

the concealing $S' = SQNUE \oplus C'$ of SQN by C' inefficient. We propose in [*Bor+19*] three main countermeasures to solve this problem with a formal analysis of their security (albeit our models have limitations since the underlying verification problem is challenging; see Section 7.4.3). Note that, when discussing our countermeasures, we also considered practical aspects related to the existing and next-generation cellular networks.

In particular, our first fix we propose is based on the idea of using symmetric encryption, that old SIM cards are already capable of, to protect *SQN* instead of the current concealing mechanism based on XOR. This fix can be deployed in new and old cellular systems but require new SIM card software. We also discuss in [*Bor+19*] how this fix also provides an opportunity to address the known linkability attack based on failure messages discussed in Section 7.4.2. Indeed, it suffices to hide the reason for the failure in the plaintext of the encryption. Our second fix is based on using subscriber-generated randomness in the KDF deriving the conceal factor C' to protect *SQN*, which is suitable for next-generation networks where USIMs are capable of generating good randomness. In our third fix, USIM protects *SQN* using asymmetric encryption, which is also suitable for next-generation networks. As we shall see in Section 7.4.4, 3GPP ended up implementing an alternative fix similar to our second fix but uses SQN_{UE} instead of fresh randomness generated by the USIM to derive C' in 5G AKA.

On the Role of *SQN* The purpose of the *SQN* counters is to provide replay protection for the subscribers. This mechanism was introduced in 3G in 2000, when the USIM was incapable of generating good randomness. This is no longer the case in 5G, where USIMs can perform randomized asymmetric encryption (required to compute *SUCI* from *SUPI*). Therefore, authentication protocols should be rethought and more standard challenge-response mechanisms could be used to replace the *SQN* counters. This would benefit the current authentication methods, which can suffer from de-synchronization and must keep the privacy sensitive *SQNs* up-to-date and sometimes fail to protect them against attackers (see Section 7.4.2).

Fixing WifI-based privacy attacks We provide in [*OBH17*] a number of countermeasures to our WiFi-based attacks that may be deployed to mitigate these attacks. One of these is the encryption of subscriber's identities over the air interface, which was already planned at the time and that is now implemented in 5G. We also document other countermeasures and mitigations that could be deployed right away by the operators, or vendors and mobile OS manufacturers. One of these is to support and enable the protection of identity privacy in the form of the so-called *conservative peer mode* (*i.e., UE* refuses to send permanent identifier if temporary identifier exists) and the use of pseudonyms [*OBH17*]. As a result of our responsible disclosure on these issues to Apple, they developed support for conservative peer mode in iOS10.

7.4.3. Challenges with Formal Analysis of Privacy Properties for AKA Protocols

The limitations of our [*Bas+18*; *Bor+19*; *OBH17*] and others' formal analyses of AKA privacy in the symbolic model [CD19; WZX21; Hus+19; EAL20][5] can be explained by the fact that the AKA protocol and its features cause several difficulties to the state-of-the-art tools and methods

5: Note that there have been analysis in other models: *e.g.,* game-based proofs in the computational model [FOR16] or machine-checked proofs in the Bana-Comon (BC) model [Kou19]. Although they provide stronger guarantees as they model more faithfully cryptographic primitives, they also require significant manual effort and expertise to be applied to complex protocols such as AKA. Moreover, they usually have a narrower scope than symbolic models analyses in terms of varieties of threat models and properties and in protocol features that can be modeled. See Section 2.4 for a discussion on the different models for formal verification of security protocols.

(such as ProVerif [Bla01] and Tamarin [Mei+13]): (i) the modeling of the $\oplus$ operator that most tools cannot handle at all (with the notable exception of Tamarin), (ii) the presence of a (non-monotonous state) state (*i.e., SQN* whose value must be stored from one session to the other), and more importantly, (iii) basic arithmetic (*i.e., SQN* is basically an integer and integer additions and comparisons are carried out by the USIM). Related to (iii) is the fundamental modeling challenge of ensuring that the equational theory is sufficiently expressive. In particular, abstracting integer operations—as typically done in symbolic models—would cause the attack to be missed, since $(X+1) \oplus X$ would then be indistinguishable from a random value. In addition, AKA protocols involve three parties (*UE*, *SN*, and *HN*) and rather complex trust relationships between them, which makes the modeling and analysis more complex. Moreover, location privacy or unlinkability (sometimes called untraceability) are no reachability properties but observational equivalence-based properties that are notoriously more difficult to verify (see Chapter 4).

More fundamentally, there are challenges related to identifying and defining privacy goals. Our subscriber activity monitoring attack based on the leakage of *SQN* confidentiality is an interesting illustration. Indeed, neither the standard nor prior privacy analyses of AKA protocols considered secrecy of this *SQN* counter as a privacy goal. To avoid such mistakes in the future, an interesting research direction would be to shifting from "verifying privacy goals defined a priori" to "verifying the absence of any (symbolic) privacy leak compared to some acceptable leak model"; in the spirit of emulation-based proof techniques based on ideal functionality (*e.g.,* Universal Composability (UC) [Can01]) or the α/β-privacy approach [MV19].

Finally, another challenge well-illustrated by our subscriber activity monitoring attack is the difficulty to identify and model a variety of realistic threat models. Focusing on a single, overly powerful adversary, *i.e.,* active and ubiquitous (controlling the wireless network all the time) adversary, would yield the expected privacy attack leaking the number of AKA sessions of all *UEs*, shadowing the more subtle and realistic subscriber activity monitoring attack and threat model we considered, *i.e.,* "sporadic attacker" (active but only in limited areas). At the other extreme, considering only passive attackers would miss the attack altogether and conclude that 5G AKA is fully privacy-preserving. The reality lies in between those two extremes, and identifying and modeling such a variety of realistic threat models is a challenge. Ideally, one would like to have a formal framework to define and compare such threat models and the trade-offs between threat models and privacy guarantees, very much the same way we use lattices of security goals to define and compare confidentiality and authentication properties (see for example Chapter 5).

In summary, state-of-the-art mechanized formal verification does not provide an off-the-shelf technique to verify privacy-related properties on the AKA protocol with sufficient precision. Considering the importance and ubiquity of the AKA protocol, we think that formally verifying it and solving the aforementioned challenges in order to provide a precise symbolic model of the AKA protocol and its variants is a substantial but major goal. We revisit these challenges and research directions in Section 11.1.

7.4.4. Responsible Disclosure and Further Impact

For all the different works presented in this Chapter, we have contacted all stakeholders to responsibly disclose the weaknesses and attacks we have

found, as well as our recommendations and fixes. We now summarize the timeline of our disclosures and their impact:

2016 Responsible disclosure of our WiFi-based privacy attacks [*OBH17*] to GSMA and device manufacturers (Google, Apple, Microsoft, Blackberry).[6] *Apple then implemented a countermeasure in iOS 10 (conservative peer mode).*

May 2017 Publication of our paper presenting our Wifi-based privacy attacks at the international workshop [*OBH17*], the latter received the *best paper award.*

Early 2017 Responsible disclosure of our *SQN* confidentiality attack [*Bor+19*] to 3GPP and GSMA. We later presented our attack at the GSMA Security Group meeting in June 2017 and again in April 2019.

March 2017 We gave a presentation of our *SQN* confidentiality attack [*Bor+19*] at the top-tier hacking TelCo conference **Troopers** [*BH17*].

July 2017 A co-author gave a presentation of our *SQN* confidentiality attack [*Bor+19*] at the top-tier hacking conference **Black Hat USA** [*Bor+17*].

2018-2022 Various vendors have proposed to 3GPP different Change Requests (CR) to fix our attack in 5G: Qualcomm, Gemalto, China Mobile, Thales, Nokia, ZTE, and Huawei, *etc.*

October 2018 Publication of our paper presenting our 5G AKA formal analysis at the ACM CCS conference [*Bas+18*].

Fall 2018 We got included in the **GSMA Hall of Fame** (entry CVD-2018/0012) that "recognises and acknowledges the positive impact [we have had] on the mobile industry"; for our formal analysis of 5G AKA [*Bas+18*].

June 2019 We obtained a formal acknowledgment of our *SQN* confidentiality attack when 3GPP added our attack to the list of the current **key issues** in 5G (see Technical Report 33.846). The above vendor proposals were then studied and refined within this TR 33.846 study item on authentication enhancements, which evaluated multiple candidate solutions for *SQN* protection (symmetric encryption, concealment with SUPI, time-based *SQN* generation, and a modified re-synchronisation procedure).

July 2019 Publication of our paper presenting our *SQN* confidentiality attack at the PETS conference [*Bor+19*].

March 2022 *One of those countermeasures,* **citing and fixing our *SQN* confidentiality attack***, was integrated into the official 5G specification V17.0.0 (3GPP TS 33.102) as a new mechanism operators can enable to mitigate our attack: "Modified AKA to avoid keystream re-use during re-synchronisations".* Concretely, the study concluded with the approval of CR 0282 (Using M' as a freshness parameter in the calculation of C') at SA3#106-e, which added Annex J to TS 33.102.[7] Annex J specifies a modified KDF_6 function that takes M' as an additional input, ensuring that the concealment value C' is unique for each SQN_{UE} value and thus preventing the keystream reuse that our attack exploited. It is a home-network decision to include M' as input to the C computation. This fix was subsequently carried forward and its naming clarified in later releases (CR 0285 and CR 0286 for Releases 18 and 19).

6: Note that we did not lead the disclosure of these attacks; Piers O'Hanlon at the University of Oxford did.

7: See Figure 7.3 for the naming of the functions and values involved in this fix.

Ethical Considerations Our research reveals weaknesses in the AKA protocol specification which is implemented in every USIM installed in 3G and 4G devices worldwide. We had followed responsible disclosure principles throughout our research and contacted all relevant parties, including 3GPP, GSMA. We conducted passive attacks only against test

USIM cards and smartphones we owned. We operated our rogue 4G base station inside a Faraday cage [Gam], to comply with legal requirements.

Media Coverage Our privacy attacks got some press coverage: ZDnet 1 and 2, The Register 1 and 2, Forbes, EFF, International Business Times, NextInpact, Silicon.de. Our formal analysis of 5G AKA also got some press coverage: Daily Mail, The National, ACM TechNews, The Hill, SRF 4, Tages-Anzeiger (frontpage), ETHZ news (frontpage), The Courier, 20 Minuten. We also wrote popular science articles about the security flaws in mobile communication protocols, in ERCIM News [*HSD19*] and in the Binaire magazine of the French Society of Computer Science (Société Informatique de France).

Reflecting on the Standardization Process As shown in the above timeline, the standardization process took several years to finally standardize a fix to our *SQN* confidentiality attack. Note also that we never received any feedback from 3GPP or GSMA after the presentations we gave for them in 2017 and 2019. Discussions were happening behind closed doors between vendors and operators, and we were not part of them. Incidentally, it was by sheer luck that we found out that they finally standardized a fix in 2022, through manually monitoring the change requests and specification versions online.

Regarding our formal analysis of 5G AKA, we also never received any feedback from 3GPP or GSMA after being included in the GSMA Hall of Fame in 2018. One major vendor responded in private communication that, albeit interesting, our analysis and recommendations to improve the specification are lacking implicit assumptions that are known to the standardization community though not explicitly stated in the standard.

> The researchers suggest standards to state more formal requirements, since this would make it easier to verify security properties. While the latter is true, it must also be understood that telecom standards are primarily intended to ensure interoperability and consensus between the contributors to secure that the networks, devices, and so on work together, rather than focusing on product specifications. Many requirements and assumptions are implicitly assumed though by tradition in the standardization community.

We answered:

> We understand that this has been the 3GPP standardization process so far. However, by not making such crucial security aspects publicly available, 3GPP risks to lose the interest of experts outside 3GPP, such as academic researchers. [...] We hope 3GPP will be able to open its standardization process and rely on less implicit assumptions and goals. Again, we think that much benefits could come out of such a paradigm shift, as witnessed by past experiences [PM16; Bas+15].

Years after, almost a thousand works have cited [*Bor+19*; *Bas+18*] to further analyze AKA protocols. We also believe that our model of 5G AKA provides a valuable tool to accompany the 5G and upcoming 6G standard's evolution and assess the security of future proposal updates. Incidentally, our Tamarin models have been independently extended to explore AKA protocol variants and their security properties [Li+25; WZX21]. Our model can also serve as the basis for a comprehensive formal comparison between AKA protocols from all generations, providing precise answers

to questions like “what guarantees does one obtain, or lose, when moving from 4G to 5G?”.

That said, our past and ongoing experiences with 3GPP and GSMA have somewhat dampened our motivation to further contribute to the standardization process. Regarding privacy, the verification challenges discussed in Section 7.4.3 have not been addressed as far as we know. We are still convinced that these constitute interesting and challenging research problems that are worth investigating for the research community.

Breaking and Fixing E-voting and French National Election 8.

🔍 *Focus on "Reversing, Breaking, and Fixing the French Legislative Election E-Voting Protocol" [DH23]*

This Chapter is primarily based on our paper [*DH23*], with additional material on the further impacts this work had on the French e-voting ecosystem (Section 8.5) as well as brief summaries of our other past contributions related to e-voting security (Section 8.6) and a general conclusion (Section 8.8).

8.1 Introduction 122
8.2 Context 124
8.2.1 Architecture 124
8.2.2 Security Goals and Threat Model 125
8.3 Reverse the Protocol . 127
8.4 Vulnerabilities, Attacks, and Fixes 130
8.4.1 Vulnerabilities 130
8.4.2 Attacking and Fixing Verifiability 132
8.4.3 Attacking and Fixing Ballot Privacy 134
8.5 Recommendations and Lessons Learned 139
8.6 Breaking and Fixing Other E-Voting Components 141
8.7 Responsible Disclosure and Further Impact . . 142
8.8 Conclusion 145

8.1. Introduction

Verifiability is a central goal in e-voting: it allows voters and auditors to verify the election result. Many e-voting protocols achieve (individual) verifiability for the voters thanks to a receipt bound to their ballots (*e.g.,* [CGG19; JCJ05; CCM08; Adi08]). Receipts allow voters to track the presence of ballots in the final bulletin board and election result and prevent a compromised or malicious voting server from dropping or tampering with their cast ballots. In this work, we ask the question how the French Legislative E-Voting Protocol (FLEP) does so by conducting a comprehensive security analysis.

The FLEP was the e-voting protocol used to organize the French legislative election for French residents overseas in June 2022 with 1.6 million eligible voters. This was the largest election (in terms of expressed votes) worldwide using e-voting. In total, to elect 11 deputies, more than 524k ballot have been cast and tallied[1]. The voters massively preferred using the FLEP (76%) over traditional paper-based voting (22.7%) or postal voting (0.3%) [flep22-r1].

As expected, the FLEP has high security ambitions. Some of those ambitions are related to lawful requirements and recommendations from all relevant regulatory bodies. To meet those requirements, notably that it remains secure under strong threats such as internal threats, the FLEP has undergone audits and the organizers have put in place an external third-party providing verification services operated by independent French researchers. Voters were encouraged to visit this web-service to verify the presence of their ballot (and receipt) in the ballot-box. For the researchers to independently develop such a tool, the vendor made public a rather incomplete specification partially describing the verifiability mechanisms used in this protocol.

In the following sections, we answer the following question:

> *Does the FLEP meet its goals? Was the 2022 French national election using this protocol secure?*

We reverse-engineered the FLEP and found flaws in the protocol and its implementation that could have been exploited to stealthily defeat ballot privacy of target voters and verifiability under a *voting server attacker* (compromised voting server) or under an even weaker *channel attacker*, that is: honest voting client and compromised plaintext channel

1: This number includes the first and the second round of the election. To give a comparison, the second largest such election is the 2015 Australian state election with 280k expressed votes using iVote, that is 6% of the expressed votes, to elect 93 deputies.

client-server (an example of concrete scenario is a compromise of the voting server certificate). That is a strictly weaker attacker model than the standard compromised server threat model (which can be the result of internal threats); see Table 5.2.

Contributions. We contribute the following.

Reverse. We reverse engineered the obfuscated JavaScript program running in the FLEP voting clients. Doing this, fully passively in order to not alter the election, on a first version used during a large-scale test election and a second version during the main election, we were able to cross-reference information in order to fill the several critical gaps in the partial specification of the FLEP. We obtained this way a full specification of the voting client and the verifiability checks. We also studied the French lawful requirements and relevant recommendations to define a precise threat model specification that we argue is in line with the French legal framework and is also supported by the literature.

New Vulnerabilities. Analyzing this, we found two vulnerabilities that can be exploited by a channel attacker and even more so by a voting server attacker, which are both included in our threat model:

- (V_1) A channel attacker can break the bound between voters' ballots and their receipts due to an implementation flaw in the voting client.
- (V_2) We found that the sub-election identifier (associated to a consulate) is not correctly cryptographically bound to the ballot, but is to the receipt. Combined with (V_1), this allows a channel attacker to modify the sub-election identifier of a ballot.

Attacks. We show how a channel attacker could have exploited those vulnerabilities to stealthily carry out the following attacks[2] (that is without leaving any evidence of the attacks):

2: We also discuss 2 other attack variants.

- (A_1) By providing crafted receipts to voters, a channel attacker can selectively drop voters' ballots to defeat individual verifiability and modify the result of the election.
- (A_2) Choose the ballot that will be cast in place of the genuinely sent ballot while still providing a good looking and valid receipt. This attack is a variant of A_1 and is more effective at modifying the election result.
- (A_3) Defeat ballot privacy of target voters. We show how a channel attacker can learn how *target* voter(s) voted by moving around ballots from sub-election (*i.e.,* consulate) to another and observing the per-consulate result. We stress that this can be done while evading all possible detection and only assuming a channel attacker; in particular decryption trustees can all be trustworthy.

We do not claim that such attacks happened. We solely claim that a malicious or compromised channel or voting server had the technical ability to perform such attacks without leaving any evidence. We responsibly disclosed those attacks to all impacted and involved actors: the operator Europe and Foreign Affairs French Ministry (EFA Ministry), its institutional security advisor Agence nationale de la sécurité des systèmes d'information (ANSSI) [3], and the vendor Voxaly Docaposte. All of those 3 stakeholders have acknowledged and confirmed our attacks. We later disclosed our findings to the 3rd-party services supervisors.

3: The ANSSI is the *French National Agency for the Security of Information Systems* whose missions include cyber defense of state information systems and to provide advice and support to government and operators of critical national infrastructure.

Fixes. We propose and discuss 5 countermeasures to those attacks. The EFA Ministry, ANSSI, and the vendor (Voxaly Docaposte) have confirmed to us that they have used some of our countermeasures to fix the FLEP. Already for the new elections that were organized one year later in June

2023 (because the election was contested in some constituencies), FLEP has been partially fixed thanks to our work, as witnessed by the new specification [Doc23]. We do not claim the absence of other attacks but only that the attacks we found were fixed.

Lessons. Designing a secure e-voting system is notoriously difficult. Flaws are regularly discovered on real-world protocols; *e.g.,* on those used in Estonia [Mül22], Switzerland [Hai+20], Russia [GG20]. In the case of the FLEP, the protocol is claimed to be derived from the state-of-the-art academic protocol Belenios that is proven secure [CGG19] (for the threat model we consider here) and is affected by none of those attacks. What went wrong?

To answer, we draw more general conclusions and lessons from this instructive experience where an academic protocol meets the real-world, challenging constraints of a large-scale and political election. Some of those translate to new open research questions. Those lessons are of a broader interest. In particular, we highlight the pitfalls in which the deployment of the FLEP fell, that are of general interest as they could have impacted the deployment of other state-of-the art protocols.

Outline. In Section 8.2, we present the architecture, the security goals, and the threat model for the FLEP. In Section 8.3, we explain how we reverse-engineered the FLEP to build a system and threat model specification and describe the latter. In Section 8.4, we present the vulnerabilities and attacks we found and discuss our fixes. In Section 8.5, we draw more general lessons from this work, that go beyond the FLEP and discuss our responsible disclosure process with the relevant stakeholders and the further impacts of this work. Finally, we conclude in Section 8.8 after having briefly summarized our other past contributions related to e-voting security in Section 8.6 and discuss in Section 8.7 our responsible disclosure process with the relevant stakeholders and the further impacts this work had.

8.2. Context

We first describe the context of the FLEP, its security goals, and threat model.

8.2.1. Architecture

The specification [Doc22] published by the vendor is largely incomplete but provides useful information about the architecture and how the FLEP is deployed in practice.

Geographical Organization. French citizens overseas are gathered in 11 electoral regions (*e.g.,* north America, north Africa, *etc.*), which are *constituencies*. Each of the constituencies is itself split into several consular sub-regions which are *consulates* and typically represent a country or a city. Each constituency elects one deputy using the FLEP with a unique election identifier electionId cryptographically bound to each ballot. Each of those elections are organized in two rounds with their own identifier roundId $\in \{1, 2\}$. Additionally, the French law requires that the results are also published at the level of consulates. Therefore, to each consulate in a consistency is associated a ballot-box with a unique identifier ballotBoxId. Figure 8.1 sums-up the organization.

Protocol Roles. Similarly to state-of-the art protocols such as He-

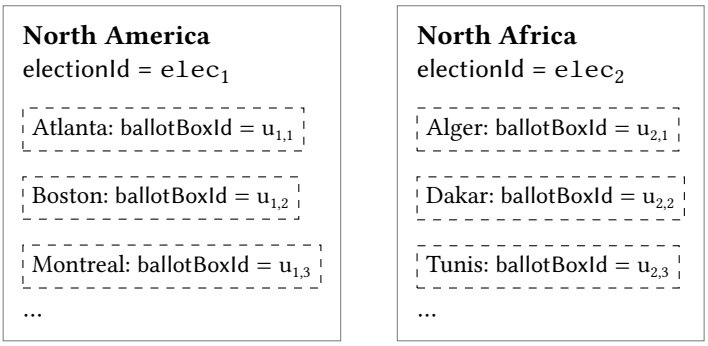


Figure 8.1.: Organization of the French legislative elections (partial view). Each constituency (*e.g.,* North Africa) runs its own election to elect one deputy. The eligible voters of each constituency are grouped into consulates (*e.g.,* Alger) with their own ballot-box. Per-consulate results are also published.

lios [Adi08] or Belenios [CGG19], the FLEP relies on different *roles* detailed next.

- **Voter:** they are the agents who will cast a vote. The FLEP assumes that voters own an email address and a cellphone number to receive login/password and confirmation codes before and during the election.
- **Voting device:** it is the device used by the voter to create and cast their vote. In the FLEP, the voting device is a JavaScript program provided by the voting server and executed in the voter's browser.
- **Voting server:** it is a server operated by the EFA Ministry whose purpose is to authenticate voters and collect all the ballots.
- **Decryption authorities:** they are the authorities who can decrypt the ballots. More precisely in the FLEP, they are 8 couples holder/deputy, and each of these 16 authorities own a share of a decryption key based on a 4-threshold encryption scheme, that is only a group of at least 4 authorities can collude to decrypt.[4] The 16 decryption authorities generate a unique public encryption key pk_E for all constituencies and their corresponding private keys $\mathsf{sk}_{\mathsf{E}1}, \ldots, \mathsf{sk}_{\mathsf{E}16}$.
- **3rd-party**: it is an external server operated by independent researchers and engineers from a French lab (LORIA) and research institutes (CNRS, INRIA), commissioned by the EFA Ministry. It provides to the voters some verification web-services [CGGa] whose purpose is to ensure the integrity of the election. This independently developed and open-source program is available at [CGGb].

4: We will come back in Section 8.5 on this threshold and how the legal requirements for decryption could be better reflected in the cryptography.

8.2.2. Security Goals and Threat Model

To conduct a security analysis of the FLEP, one first needs precise security objectives and threat models. While the partial specification [Doc22] fails to do so, there fortunately exist lawful requirements that the FLEP should comply with, notably the Code électoral [éle] (*i.e.,* the French law governing elections) and the recommendations enacted by the Commission nationale de l'informatique et des libertés (CNIL) [5] [Off19], which are a list of requirements that such a system is expected to meet (even if there is no legal obligation).

5: The CNIL (*National Commission on Informatics and Liberty* in English) is an independent French administrative regulatory body whose mission is to ensure that data privacy law is applied.

We have studied this legal framework [éle; Off19] and derived security goals the FLEP should achieve and under which threat model. Our list of objectives is not exhaustive but we focus here on those that are relevant for our work. We refer the reader to [*DH22*, Appendix A] for detailed explanations about our study of this framework and how we derived the security goals and threat models that we summarize next.

Ballot Privacy. The Code électoral [éle, R176-3-9] and the CNIL require-

ments [Off19, SO 1-04, 1–07] agree on the fact that the FLEP must ensure the *confidentiality of the votes*.

Verifiability. Second, the FLEP must ensure the integrity of the election outcome and notably **individual verifiability** [éle, R176-3-9],[Off19, SO 2-07, 3–02], that is each voter should be able to verify that their ballot has been added into the ballot-box; *i.e.,* it was not dropped or tampered with.

For instance, the requirement [éle, R176-3-9] requires that "the voter is provided with a digital receipt allowing them to verify online that their vote has been taken into account". As we shall see, the FLEP uses some hashed values over the cast ballot and some meta-data as well as a signature as *digital receipt*. The voter can verify online this receipt at the voting server or the 3rd-party server, the latter independent 3rd-party verification option must exist to comply with [Off19, SO 3-02].

Threat Model. We shall define a threat model that we argue is in line with the legal framework of the FLEP. As detailed in [*DH23*, §A], we derive from the CNIL recommendations [Off19, Security level 3, SO 3-02] that the FLEP must guarantee the integrity of the election even under a compromised voting server, which also motivates the requirement to rely on an independent, 3rd-party verification service. Most of the attacks we shall present actually make weaker assumptions, *i.e.,* assume an even weaker attacker. Indeed, instead of considering a (possibly) compromised voting server, we shall assume a compromised communication plaintext channel between the voters and the server, we call such an attacker a *channel attacker*. More precisely, the *channel attacker* can intercept and inject (plaintext) messages in-between voters under attack and the voting server but has not necessarily access to the voting server internals, such as its databases, the signing sheet, the authentication material, the log files, *etc.*

Table 5.2 from the chapeau of this part summarizes the assumed trustworthiness of all roles for the different properties (the less trustworthy parties there are, better is the protocol). It also compares this with the threat model under which the Belenios protocol (and other state-of-the-art protocols such as Helios [Adi08]) were proven secure. (An honest public bulletin board is usually assumed, which can be achieved by an honest 3rd-party for comparison.) It shows that our attacks break the FLEP under an (even weaker) attacker than the standard one for which Belenios was proven secure [CGG19], despite FLEP being derived from Belenios [Doc22].

Remark 8.2.2 Assuming the voting client as a trustworthy component can be considered too strong an assumption, even more so due to the latter being a JavaScript program served by the voting server and executed in the voter's browser. Some e-voting systems such as the IVXV [Hei+16], Helios [Adi08], or the Swiss Post protocol [Swi], aim at ensuring the well-known *cast-as-intended* property, that is ensuring individual verifiability under a compromised voting device that could try to modify the intended vote. The FLEP protocol has obviously not been designed to protect the voters against such a threat. Therefore, to conduct a fair security analysis, we decided to assume the voting client as a trustworthy component. We discuss in Section 8.5 how this assumption can be made realistic in practice with the notion of "*universal integrity checks*" where anyone can check the JavaScript integrity to detect malicious voting servers serving malicious code. ▷

Remark 8.2.1 (Examples of channel attackers) A channel attacker can obviously be realized by compromising the voting server. Indeed, even if the converse is wrong in general, a voting server attacker is also a channel attacker. More interestingly, it can be realized by compromising parts of the network server infrastructure. Specific operational countermeasures are usually deployed to protect the critically important server(s) on which the election runs. However, the server(s) are connected to the Internet through a more complex and shared network infrastructure which may appear as an Achilles heel. The TLS certificates shared across the infrastructure might be less protected and shared across more devices such as TLS middle-boxes that could be deployed to monitor the plaintext traffic to mitigate *e.g.,* DDoS attacks. This, unfortunately standard, solution would undermine the security of the communication channels voters-voting server by increasing the attack surface and the risk of *MIM* attacks yielding a channel attacker.
Finally, a channel attacker could also be the result of a compromise of the network infrastructure on the voter side. For similar reasons, a voter's company may monitor the plaintext Internet traffic to protect its employees and its assets. Unfortunately, this would, again, introduce a *MIM* in the communication channel between the voter and the server. Voters have not been made aware of this threat in the context of this election.
Therefore, a channel attacker is an interesting threat model in itself as it is strictly weaker than the server attacker and can be realistic in some scenarios. ▷

Table 8.1.: Main cryptographic values involved in the FLEP.

Name	Expected value
Ballot	$b := (\{v\}_{\mathsf{pk_E}}, \pi)$
Hash value	$h := \mathsf{H}(b \Vert \mathsf{roundId} \Vert \mathsf{electionId} \Vert \mathsf{ballotBoxId})$
Ballot reference	$H := \mathsf{roundId} \Vert \mathsf{electionId} \Vert h$
Activation hash	$h_{\mathrm{code}} := \mathsf{H}(b, \mathsf{code}_{\mathsf{activ}})$
Ballot fingerprint	$hb := \mathsf{H}(b)$
Seal	$\mathsf{cSU} := \mathsf{infoSU} \Vert \mathsf{sign}_{\mathsf{sk_S}}(\mathsf{H}(\mathsf{infoSU})) \Vert \mathsf{pk_S}$
	$\mathsf{infoSU} := \mathsf{roundId} \Vert \mathsf{electionId} \Vert \mathsf{electionName} \Vert \mathsf{ballotBoxId} \Vert hb^{s_5}$

8.3. Reverse the Protocol

[Doc22] only provides a partial specification, that is so incomplete that the attacks we shall present could not be even described[6]. Indeed, this document focuses on the expected format of some verifiability-related messages (such as the receipts) but omits to specify how exactly they are computed, exchanged, and which checks are performed upon reception. Moreover, a bigger picture of the protocol is completely missing.

6: Based on our discussions with the stakeholders, Voxaly Docaposte published in February 2023 a new version of the specification [Doc23] which includes a more detailed protocol specification with some countermeasures to our attacks we discuss in Section 8.4.

Reverse Methodology By reversing the JavaScript voting client code, studying and cross-referencing different sources, we built a complete description of the FLEP voting client and all its interactions with the server as well as some important server-side components (handling of errors, verifiability checks).

Specifically, a few consenting eligible voters[7] passively captured all browser-server interactions during the election, yielding 15 JavaScript files, 4 HTML files, and 4 plain data exchanges. These files were obfuscated and minimized. A key advantage came from an earlier, less-obfuscated version of the JavaScript files used during the September 2021 test campaign [flep22-test]: by reconciling the two code bases and cross-referencing function implementations, we recovered the core protocol logic (see [*DH23*, §3]). Note that even though a specification of the 3rd-party tool was missing too, we have been able to determine the checks it performed based on an informal description of the service [CGGa], open discussions with the researchers, and an inspection of the source code. All the impacted and involved actors (Voxaly Docaposte, ANSSI, EFA Ministry) have acknowledged our system specification is correct.

7: The voters who sent us the collected data, who are computer scientist colleagues, were informed about our research and about the content of those logs. They gave us their informed consent.

Reversed Specification We now present the result of our reverse: a full description of the FLEP. We shall specify all the actions executed by the voter, the voting client, and the voting server at each step. Unlike [Doc22], we also explicit how messages are exchanged and computed. Those steps are summarized in Figure 8.2.

Step 1: The voter browses to the election website URL and connects to the voting server and receives an HTML document displaying the login page. The voter authenticates themselves using a login and password they received before the election through two different channels, the login by email and the password by SMS. In addition to the authentication data, the voting client also sends to the voting server client_info (some meta-data about the voting client) and b_{test} which is a dummy ballot encrypted with a dummy election public key that serves as sanity check that the voting client will be able to correctly compute the real ballot at step 4.

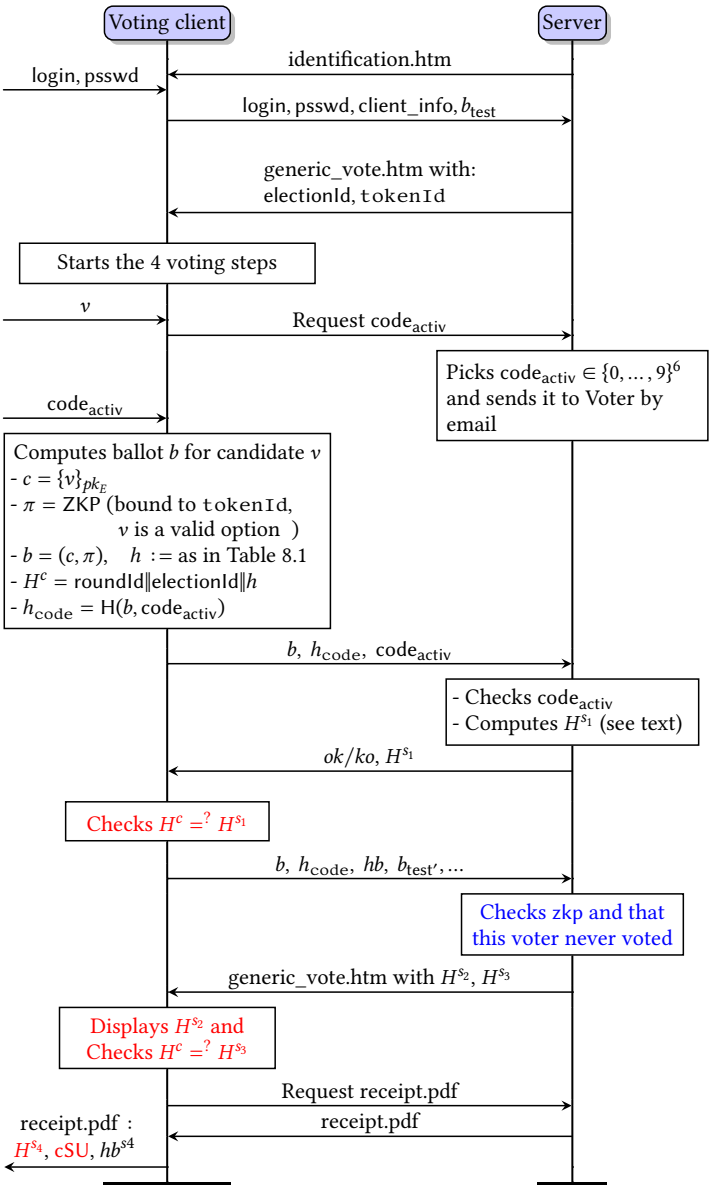


Figure 8.2.: Description of the 2022 French Legislative Election Protocol (FLEP). Arrows on the left correspond to interactions with the human voter. Cryptographic messages are specified in Table 8.1. Actions in red will be relevant for the vulnerabilities presented in Section 8.4. Actions in blue, and only those, are elements obtained during discussions with the stakeholders. A more detailed diagram is given in [*DH22*, Appendix C].

Step 2: The voting client receives an HTML document displaying the different candidates $\mathscr{V}$ the voter can choose. This document also contains some hidden identifiers such as the election identifier electionId or the ballotBoxId and a per-session unique identifier `tokenId`. The voter chooses one candidate $v \in \mathscr{V}$ for whom they want to vote.

Step 3: The voter clicks to confirm their choice.

Step 4: The voting client sends a request to the voting server to generate an *activation code*, acting as a second authentication factor. The voting server generates such an activation code $\text{code}_{\text{activ}}$ (made of 6 random digits) and sends it to the voter using a side-channel (*i.e.*, by email). The voter receives this code and enters it in the voting client and clicks on the "Vote" button. The voting client then computes some cryptographic material explained next and summarized in Table 8.1.

• First, the *ballot* $b := (c, \pi)$ that is made of the encrypted vote v ($c := \{v\}_{pkE}$) with the election public key (pkE) along with a Zero-Knowledge Proof (ZKP) (π) proving that v is a legitimate choice of candidate, *i.e.,* $v \in \mathscr{V}$. This proofs is also bound to `tokenId` and electionId received at Step 2.

• The voting client also computes the *hash value* h where $\|$ is a delimiter, roundId is the round of the election (1 or 2), electionId is the election identifier, and ballotBoxId is a per-ballot-box (hence per-consulate) unique identifier.

Hash value: $h =$ H($b\|$roundId$\|$electionId$\|$ballotBoxId)

• The *ballot reference* H (omitting correction codes) is computed and exchanged at different steps of the protocol, we specifically note H^c to refer to the *ballot reference* that is computed by the voting client and stored in the SessionStorage of the voting client browser[8].

Ballot reference: $H =$ roundId$\|$electionId$\|h$

• The *activation hash value* $h_{\texttt{code}} := \mathsf{H}(b, \text{code}_{\text{activ}})$. This value was not described at all in the specification [Doc22] and was supposed to act as a proof of knowledge of the activation code $\text{code}_{\text{activ}}$, bound to the ballot.[9] That said, the precise role of $h_{\texttt{code}}$ is unimportant for the rest of the presentation.

The values b, $h_{\texttt{code}}$, and $\text{code}_{\text{activ}}$ are then sent to the voting server. The voting server then verifies the validity of the activation code $\text{code}_{\text{activ}}$, recomputes the *ballot reference* H, denoted by H^{s_1}; indeed H^c is not sent to the voting server. The response to the above request is $(ko, _)$ if the activation code was invalid and (ok, H^{s_1}) otherwise. Upon reception of the response, the voting client verifies that the response equals (ok, H^c). Therefore, H^{s_1} must be equal to H^c. This test provides the voting client evidence that it executes the protocol in the same context as the one of the voting server.

Finally, if those tests succeed, then the voting client computes the *ballot fingerprint*: $hb := \mathsf{H}(b)$. The ballot b is again sent to the voting server, along with hb and $h_{\texttt{code}}$.

At this point, the voting server verifies that the voter never voted before (revoting is forbidden in the FLEP) and that the ballot ZKP π is valid. If so, the voting server stores the ballot b in the ballot-box ballotBoxId associated to the voter (one per consulate). It also adds the voter to the *signing sheet* containing all the voters who voted so far. Finally, it computes and stores a *seal* cSU associated to this ballot, that is a signature over the ballot and some metadata that will be part of the receipt provided to the voter in the final step (we specify the seal cSU below).

Step 5: Finally, the voting client receives from the voting server an HTML document that displays the *ballot reference* H and that asks the voter to click a link to download the PDF receipt. Voters are also encouraged to click a link to verify that their ballot is indeed included in the ballot-box.

Two occurrences of the *ballot reference* H are present in this HTML document, which is generated and sent by the voting server. We thus denote those two occurrences with respectively H^{s_2} and H^{s_3}. The first occurrence H^{s_2} appears in a HTML tag $<$ `pclass = "recepisse − code"` $>$ and corresponds to the value being displayed to the voter. The second occurrence H^{s_3} appears in a JavaScript script embedded in the page that tests that H^{s_3} equals the *ballot reference* stored in the voting client SessionStorage, *i.e.,* H^c. Therefore, $H^{s_3} = H^c$ or an error message is displayed.

PDF receipt: The very last step occurs when the voter clicks the link to download the PDF receipt from the previous page, which replaces

8: The SessionStorage is a storage local to the browser and associated to the current tab. It can store session-specific data that persist across page reloads and page redirections. See, for example, the documentation for Firefox [sessStorage].

9: We note that this is completely inefficient since the activation code $\text{code}_{\text{activ}}$ is actually sent in clear text along $h_{\texttt{code}}$ (with the ballot b). According to the vendor, this was a mistake and will be fixed. Once fixed, $h_{\texttt{code}}$ provides such a proof, even though $\text{code}_{\text{activ}}$ can be quite easily brute-forced given $h_{\texttt{code}}$ and b.

the previous page (which cannot be loaded any more). The downloaded PDF receipt contains the *ballot reference H*, the seal cSU, and the *ballot fingerprint hb*. Despite those values being (partially) specified in [Doc22], the values displayed in the PDF receipt are never checked by the voting client and could differ from the expected, specified values. Therefore, we note H^{s_4} the *ballot reference* and hb^{s_4} the *ballot fingerprint* that are displayed in the PDF receipt.

The seal cSU should be computed by the voting server (already at step 4) as follows: $\mathsf{cSU} := \mathsf{infoSU} \| \sigma \| \mathsf{pk_S}$ where infoSU is defined as:

$$\mathsf{roundId} \| \mathsf{electionId} \| \mathsf{electionName} \| \mathsf{ballotBoxId} \| hb^{s_5}$$

and electionName is the name of the election, hb^{s_5} is supposed to be the *ballot fingerprint*, and σ is a digital signature with the server's signature private key $\mathsf{sk_S}$ (associated to the signing verification key $\mathsf{pk_S}$) computed as follows: $\sigma = \mathsf{sign}_{\mathsf{sk_S}}(\mathsf{H}(\mathsf{infoSU}))$.

Decryption authorities: At the end of the election, a tally ceremony involving a threshold of the decryption trustees occurs. They collaborate to decrypt an homomorphic aggregate for each ballot-box, produce ZKPs of correct decryptions, and announce the election results.

3rd-party role: The 3rd-party contributes to both the individual and universal verifiability. To this aim, the EFA Ministry sends it the results of the tally (*i.e.,* the ballot-boxes, ZKPs, and results). First, for checking universal verifiability, the 3rd-party checks the validity of the ZKPs of correct decryption. At the constituency level, it checks that those proofs are valid *w.r.t.* official results publicly published onto the EFA Ministry website. Moreover, it checks that all the ballots included in the ballot-boxes are well-formed, *i.e.,* that their ZKPs are all valid. For allowing individual verifiability checks, the 3rd-party proposes a web service to the voters who can check that the seal of their receipts contains a legitimate signature of the voting server. Moreover, once the election is over and the 3rd-party has received the ballot-boxes, the 3rd-party also verifies that the seal corresponds to a legitimate ballot[10]. After the protest period, *i.e.,* ten days following the results announcement, the 3rd-party publishes a report [CGGa] containing a summary of all their verifications.

10: In 2022, voters had to use this service once the election was over, and not before, to obtain this guarantee. In 2023, based on our recommendation (see [*DH22*, Appendix E.3.1]), the 3rd-party decided to log all the seals received during the voting phase to do this check as soon as it becomes doable.

8.4. Vulnerabilities, Attacks, and Fixes

We present the two vulnerabilities we found (Section 8.4.1). We then show how these vulnerabilities can be exploited to break verifiability and the integrity of the election (Section 8.4.2) and the confidentiality of target voters' votes (Section 8.4.3). We also propose fixes to those attacks, some of them are or will be deployed by the FLEP stakeholders. We summarize all the attacks we found in Table 8.2. We also provide the fixes chosen by ANSSI, EFA Ministry, and Voxaly Docaposte and already implemented for the elections organized in 2023 and 2024.

8.4.1. Vulnerabilities

When reversing the specification, we uncovered 2 critical vulnerabilities that could be exploited by a channel attacker (*i.e.,* an attacker controlling the plaintext communication channel) and even more so by a voting server attacker (*i.e.,* a compromised voting server). As we shall see, they impact the integrity of the election and the confidentiality of the votes.

Table 8.2.: Summary of the attacks found. The last column presents the fixes that have been chosen by the stakeholders and implemented for the elections organized in 2023.

Name	Attack on	Threat model	Impact	Fixes deployed in 2023
Replace	Indi. Verif.	Channel att.	Replace any cast ballot	Fix 1: display H^c
Drop	Indi. Verif.	Channel att.	Drop any cast ballot	
Swap_H	Ballot priv.	Voting server att., some voters collude	Learn any target voter's vote	Fix 3: add ballotBoxId to the ZKP
Swap_b	Ballot priv.	Channel att., some voters collude	Learn any target voter's vote	
Swap_{ID}	Ballot priv.	Voting server att., some voters collude	Learn any target voter's vote	Fix 3 + Fix 5: display ballotBoxId

We stress that our reversed specification detailed in Section 8.3 was necessary to identify those vulnerabilities that could not even be described within the limited framework of [Doc22].

V1: Lack of binding of receipts with their ballots. In the FLEP, the integrity of the election is guaranteed by the use of receipts: voters can send their receipt (the ballot reference H or/and the seal cSU) to a server (the voting server or the 3rd-party) to verify that their ballots have been added in the ballot-box. Moreover, the 3rd-party ensures, by verifying the decryption ZKPs, that the result of the election corresponds to the content of the ballot-boxes. Thanks to these two checks, an attacker should not be able to tamper with the cast ballots and with the election result.

A subtlety we noticed in the JavaScript code of the FLEP voting client is that the ballot references that are displayed to the voter by the voting client (H^{s_2} and H^{s_4}) for later checks are not necessarily the same as the one it computed for the ballot produced by the voting client (H^c). This is the reason why we named those references differently, even though they refer to values that are expected to be equal (*i.e.,* equal to H^c computed by the voting client) thanks to various consistency checks (shown in red in Figure 8.2).

Unfortunately, these checks are flawed in that, among the 4 references received from the voting server, H^{s_1}, H^{s_2}, H^{s_3}, H^{s_4}, they only ensure that:

$$H^{s_1} = H^c \quad \text{and} \quad H^{s_3} = H^c.$$

There is no guarantee about the values H^{s_2} and H^{s_4}, which are the only references visible by the voters, respectively in the last web page and in the PDF receipt (both are depicted in [*DH22*, Appendix C.2]). A channel attacker can take advantage of this implementation flaw to falsify the verifiability of the FLEP and thus modify the result of the election as explained in Section 8.4.2. We stress that the human voters themselves are unable to recompute the genuine, honest value of H (that is H^c) without an expert and technical knowledge. Indeed, for instance, they never learn the value of their ballot b.

V2: Malleability of ballot-box identifiers. As presented in Section 8.2.1, the FLEP is deployed in a complex environment with multiple elections and ballot-boxes. In the partial specification of the system [Doc22], Voxaly Docaposte seems to be aware of this complexity: "*The election identifier [*electionId*] [...] will be added in the [ZK] proofs associated to the ballot. This allows to detect if a ballot has been moved from a ballot-box to another*".

Unfortunately, we found out that this statement is incorrect. Indeed, the election identifier electionId does not identify a ballot-box (associated to a consulate) but only an election (associated to a constituency). The ballots

are thus cryptographically bound to an election but not to a ballot-box whose identifier is not included in the ZKP context of the ballot.

Interestingly, the ballot reference H *does* cryptographically bind the ballot to the ballot-box identifier ballotBoxId. However, as we shall see, this ballot reference can be recomputed by an attacker for a different ballotBoxId. As we shall see in Section 8.4.3, a channel attacker can use this and the previous vulnerability (V1) to move around ballots across different ballot-boxes and attack ballot privacy.

8.4.2. Attacking and Fixing Verifiability

We now present how the first vulnerability can be exploited to break verifiability and thus the integrity of the election. We also propose and discuss fixes.

8.4.2.1. Attacking Verifiability

The first vulnerability can be exploited to falsify the individual verifiability of the system under a channel attacker. An attacker who controls the communication channels can stealthily (1) drop ballots, and (2) replace them by ballots of their choice as explained in the two attack descriptions below.

Attack [Replace]: This scenario assumes a channel attacker and requires at least two voters who vote in the same consulate: Bob, who will suffer from the attack, and, Alice, an arbitrary voter. In the following, we re-use all the notations introduced in Section 8.3 and Figure 8.2. We assume that all the variables are indexed by 1 for Alice's vote, and 2 for Bob's. The attack proceeds as follows:

Step a: Alice casts her vote as expected. In this first step, the attacker executes honestly, *i.e.*, $H_1^{s_i} = H_1^c$ for all $i \in \{1, \ldots, 4\}$ and σ_1 is honestly computed. Hence, all the checks that Alice can do to be sure that her ballot b_1 has been included in the ballot-box succeed.

Step b: Bob follows the protocol but the channel attacker intercepts all messages after the creation of the ballot b_2 and computes by itself the responses that are normally sent by the voting server to the voting client as follows: $H_2^{s_1} = H_2^{s_3} = H_2^c$ but $H_2^{s_2} = H_2^{s_4} = H_1^c$ and $\sigma_2 = \sigma_1$. That is the attacker provides a receipt that refers to Alice's ballot. We do not assume that the attacker knows the signing private key sk_S to compute σ_2 since the attacker can simply replay the values obtained at Step a. The value H_2^c can be computed by the attacker since it is only made of public data (*e.g.*, electionId and ballotBoxId) or data sent by Bob, such as b_2.

Step c: Because a ballot is not cryptographically bound to a voter, the attacker can forge a ballot $b_{\mathtt{att}} = (\{v'\}_{\mathsf{pk}_E}, \mathsf{zkp}')$ for $v' \in \mathscr{V}$ of their choice and replace each occurrence of b_2 by $b_{\mathtt{att}}$ in Bob's messages towards the voting server. More precisely, the attacker sends $b_{\mathtt{att}}$, $\mathsf{H}(b_{\mathtt{att}}, \mathsf{code}_{\mathsf{activ}})$ and $\mathsf{H}(b_{\mathtt{att}})$ instead of b_2, h_{code}, and $\mathsf{H}(b_2)$. The voting server then registers the adversarially-chosen ballot $b_{\mathtt{att}}$ in the name of Bob.

Even if, at Step b, Bob receives inconsistent data (*e.g.*, $H_2^{s_2} \neq H_2^c$), these are not detected by the checks performed by the voting client and those differences are invisible to Bob himself. (This seems similar to the clash attacks of [KTV12] but our attack does not rely on a compromised voting client to make it compute a clashing ballot.) Moreover, Bob can use the web services (provided by the voting server or the 3rd-party) to be convinced that "his" ballot has been added to the ballot-box. Indeed, Bob

got Alice's receipt which corresponds to a ballot added in the ballot-box at Step a. Finally, let us explain how the attacker can make sure Bob does not detect the replacement of his ballot in the very unlikely situation where his original ballot was the only vote for a particular choice $v \in \mathscr{V}$ in this ballot-box (since the tally will reveal v got no vote). It suffices for the attacker to make sure each of the options $v \in \mathscr{V}$ gets at least one vote: either by assuming the existence (*e.g.,* large consulates) or knowing other voters casting such ballots (*e.g.,* accomplices), or by performing the above attack on $|\mathscr{V}|$ voters to make them cast all options. As a result, all the checks Bob was suggested, instructed, or able to do succeed despite his ballot b_2 has been dropped and replaced by the attacker ballot $b_{\texttt{att}}$.

Therefore, at the end of this scenario, nobody can detect the removal and replacement of Bob's ballot and yet, the election result will not include Bob's ballot b_2 and include the attacker ballot $b_{\texttt{att}}$ instead. Alice and Bob are convinced that their ballots have been counted and the voting or the 3rd-party server will always receive consistent data. Bob is cheated by the fact that the receipt he obtained actually points to Alice's ballot, which does exist in the ballot-box. Note that a channel attacker can repeat this to attack an arbitrary number of voters acting as Bob, always using the same Alice's data. (Moreover, we describe in [*DH22*, Appendix B.1] a weaker variant [Drop] of this attack that only drops Bob's ballot instead of replacing it.)

Impact: An attacker who compromises the communication channel (or the voting server) can significantly modify the outcome of the election by dropping and replacing ballots, hence falsifying the individual verifiability property.

Remark 8.4.1 Interestingly, [Replace] is no longer possible if FLEP used the ballot format of Belenios, since this includes voters' signatures. Therefore, the removal of signatures from Belenios to FLEP—as a means to comply with CNIL recommendations (ballots should be anonymous)— without compensating this loss with another security mechanism introduced a weakness in the protocol. ▷

8.4.2.2. Fixing Verifiability

We propose 2 fixes to prevent such attacks. The first one is easy to implement but makes the voter's journey and verification tasks more complex, while the second would actually simplify them but is more complex to implement.

Fix 1: A first approach to fix the verifiability attacks is to display H^c instead of H^{s_2} on the last web page of the user interface (step 5). If this was done, a conscientious voter would be able to compare this reference to the one printed in their receipt to enforce the consistency $H^{s_1} = H^{s_2} = H^{s_3} = H^{s_4} = H^c$. Finally, the voter can then use the 3rd-party web service to check the presence of their ballot in the ballot-box with confidence. This solution would fix Vulnerability 1 as well as the two verifiability attacks [Replace] and [Drop]. However, it complicates further the voter's tasks, which are already quite complex (see Section 8.5).

Fix 2: A second, better fix is to make the voting client generate the PDF receipt. To do so, the voting server would still need to send the signature σ for the voting client to compute the seal cSU. The voting client must verify the validity of this signature (with respect to pk_S) and its content before generating the PDF receipt by itself. We prefer this solution as it does not require extra voters' checks. Moreover, it makes it possible to check the signature of the seal cSU before displaying the PDF receipt, thus detecting potential forgery or voting server misbehavior as early as possible and making the voting server accountable for potential misbehavior detected later. The main drawback is that it requires to import an external library to generate PDFs or to use a static PDF file that is then dynamically filled with missing cryptographic data.

Remark 8.4.2 The stakeholders chose to implement Fix 1 for elections in June 2023. Fix 2 seems to be their ultimate solution for a future version of the system. ▷

Remark 8.4.3 (On Fix 1) The version 2 of the specification [Doc23] (from February 2023) shows that the Fix 1 was not implemented as is. We regret that the chosen implementation does not fully prevent the vulnerability V1. Indeed, H^c is now displayed on the last web page of the user interface (step 5) and the voting client now checks that $H^c = H^{s_3}$. Very surprisingly, the vendors also decided to display H^{s_3} on the same page and even to instruct the voters to visually "check that the receipt H^c [computed by the voting client] corresponds to the receipt actually inserted in the ballot-box H^{s_3}". Asking the voters to visually check $H^c = H^{s_3}$ is useless given that this test is performed by the voting client. More importantly, this gives a false sense of security as it gives the impression no further checks are needed. The voters must actually check that the displayed receipt (H^c) corresponds to the one printed on the PDF receipt (H^{s_4}) and then use it to perform the usual individual verifiability check, preferably at the 3rd-party website.

In conclusion, the implemented fix allows a very cautious voter to detect our attacks (by comparing H^c and H^{s_4}). However, it also gives non-necessary additional tasks to the voters and fails to instruct them to perform the only useful check. ▷

8.4.3. Attacking and Fixing Ballot Privacy

The vulnerabilities (V1) and (V2) can be jointly exploited to break ballot privacy for target voters: *i.e.*, voters the attacker decides to target and attack when they cast their ballot. We shall see that a channel attacker can learn how a target voter voted provided that there are at least 2 ballot-boxes in the target voter's constituency (this is the case for all the French constituencies). We present the attack core ideas in a simplified setting before presenting the actual attacks.

Attacks overview. Let us assume a channel attacker willing to learn the vote of Alice, who is registered in ballot-box u_1 (u_1 is a ballot-box identifier ballotBoxId). The key ingredient for this attack is *inter-ballot-box move*: the attacker can cast Alice's ballot in $\mathrm{u}_2 \neq \mathrm{u}_1$ without anyone noticing. *One* approach, detailed in [Move$_H$] below, to achieve this is for the attacker to:

- move Alice's ballot b_A to u_2 and, thanks to (V1), give her a forged receipt corresponding to b_A as if it was cast in u_1.
- to preserve the number of ballots in each ballot-box, the ballot b_B of a voter eligible in u_1, say Bob, can be moved to u_1 using the same technique.

Finally, because of (V1) Alice's and Bob's receipts pass individual checks, and because of (V2), the ballot-boxes u_1 and u_2 will be tallied without raising any error, despite the move of b_A to u_2.

Now, let us assume the attacker knows a ballot-box u_2 with a few eligible voters. Intuitively, the attack is as follows: the attacker exploits the inter-ballot-box ballot move attack to cast Alice's ballot b_A in the ballot-box u_2. Exploiting the attack [Replace] the attacker replaces all other ballots cast in u_2 by ballots $b_1, \ldots, b_n$ whose values are known. Because the ballot-boxes are individually tallied, the attacker will learn the result for u_2 from which the Alice's vote encrypted in b_A can be deduced (since the

votes encrypted in $b_1, \ldots, b_n$ are assumed to be known). We present below concrete attacks exploiting these ideas and explain how the attacker can completely evade detection. We then present fixes.

8.4.3.1. Attacking Ballot Privacy

As informally described above, our ballot privacy attacks rely on two ingredients: inter-ballot-box move, that is how to stealthily move a ballot from a ballot-box to another, and how to exploit the latter to introduce the privacy leak through tallying. We first present three inter-ballot-box ballot move techniques and then explain how any one of those can be exploited to mount ballot privacy attacks.

For each of those three inter-ballot-box move techniques, we assume that the election (constituency) identifier `elec` contains at least two ballot-boxes (consulates) identifiers u_1 and u_2. Alice is eligible to vote in consulate u_1 and is the target voter whose vote will be moved from u_1 to u_2. To preserve the right number of ballots in each ballot-box, an arbitrary ballot (say Bob's) cast in u_2 is moved to u_1 (and possibly replaced).

[Move$_H$] Inter-ballot-box ballot move exploiting (V1,V2) for a voting server attacker. This first technique assumes a voting server attacker, *i.e.,* the voting server is compromised. The attack scenario is as follows:

Step a: Alice votes and sends a ballot b_A.

Step b: The voting server attacker receives b_A but stores it in the ballot-box u_2, while Alice was eligible in u_1.

Step c: Additionally, the voting server attacker computes malicious receipts that will be valid for b_A, even though it was placed in the wrong u_2:

$$
\begin{aligned}
&H^{s_2} = H^{s_4} = m \| \mathsf{H}(b_A \| m \| \mathrm{u}_2) \\
&H^{s_1} = H^{s_3} = H^c = m \| \mathsf{H}(b_A \| m \| \mathrm{u}_1) \\
&\mathsf{cSU} = m' \| \mathrm{u}_1 \| \mathsf{H}(b_A) \| \mathsf{sign}_{\mathsf{sk}_S}(\mathsf{H}(m' \| \mathrm{u}_2 \| \mathsf{H}(b_A)))
\end{aligned}
$$

where $m' = \mathsf{roundId} \| \mathsf{electionId} \| \mathsf{electionName}$ and $m = \mathsf{roundId} \| \mathsf{electionId}$ are election meta-data. Note that H^c, H^{s_1}, and H^{s_3} are computed as expected but H^{s_2} and H^{s_4} are modified such that the receipt is valid for b_A cast in u_2. This is crucial to make all 3rd-party checks succeed and avoid detection (we come back to this in Section 8.4.3.2).

Step d: Steps a, b, and c are applied to move a ballot b_B of an arbitrary voter Bob eligible in u_2 from u_2 to u_1.

[Move$_b$] Inter-ballot-box ballot move exploiting (V1,V2) for a channel attacker. In this scenario, we assume that the attacker only controls the (plaintext) communication channel.

The attack scenario is as follows:

Step a: Alice votes and sends a ballot b_A.

Step b: The channel attacker intercepts the ballot b_A and replaces it by a new ballot b_{att} they choose. In addition, exploiting (V1) (attack [Replace]), the attacker modifies the messages sent by the voting server to make Alice get a valid receipt for b_{att}, while she intended to cast b_A instead:

$$
\begin{aligned}
&H_1^{s_2} = H_1^{s_4} = m \| \mathsf{H}(b_{att} \| m \| \mathrm{u}_1) \\
&H_1^{s_1} = H_1^{s_3} = H_1^c = m \| \mathsf{H}(b_A \| m \| \mathrm{u}_1) \\
&\mathsf{cSU}_1 = m' \| \mathrm{u}_1 \| \mathsf{H}(b_{att}) \| \mathsf{sign}_{\mathsf{sk}_S}(\mathsf{H}(m' \| \mathrm{u}_1 \| \mathsf{H}(b_{att})))
\end{aligned}
$$

Note that H^c, H^{s_1}, H^{s_3} are computed as expected but H^{s_2} and H^{s_4} are modified such that they will be valid against a ballot-box u_1 containing b_{att} instead of b_A. In particular, cSU_1 is exactly the seal returned by the voting server when it received b_{att}; the attacker does not need to forge a signature.

Step c: Similarly, the attacker intercepts b_B a ballot sent by Bob, an eligible voter in u_2, and replaces it with Alice's ballot b_A exploiting (V1) (attack [Replace]). Again the attacker modifies the references H_2 and the seal cSU_2 sent to Bob replacing b_B by b_A.

[Move$_{ID}$] Inter-ballot-box ballot move without exploiting (V1) for a voting server attacker. We assume a server attacker that does not exploit (V1). Because ballotBoxId is sent by the voting server to the voting client, the server can send u_2 instead of u_1 to make the Alice's voting client compute a ballot for u_2 by itself. The displayed data and the individual checks do not allow Alice to detect that the ballot was computed for the wrong ballotBoxId. Again, to preserve the number of ballots in each ballot-box, we assume that the voting server sends u_1 instead of u_2 to Bob, an arbitrary eligible voter in u_2 so that Bob computes and casts a ballot for u_1.

[Swap$_X$] Exploiting inter-ballot-box ballot move to attack ballot privacy. We now explain how any of the three move techniques can be exploited to violate ballot privacy with three attack variants: [Swap$_H$], [Swap$_b$], and [Swap$_{ID}$]. In the following, we specify the attack [Swap$_X$] for $X \in \{H, b, ID\}$.

We already described in the *attacks overview* paragraph (section 8.4.3) the main idea of gathering Alice's ballot with ballots whose votes are known to the attacker in a single ballot-box u_2 so that when tallied, it reveals Alice's vote. In order to make this completely stealthily, the attacker needs to take some precautions.

First, the attacker needs to preserve a perfect correspondence between the number of ballots in each ballot-box and the number of voters eligible in this ballot-box who did cast a ballot. Indeed, any voter who casts a vote is registered in a semi-private signing sheet. We explain in [*DH22*, Remark 11] that it is unlikely that voters check the signing sheet due to various practical constraints. Even if they did, the attack [Move$_X$] preserves a consistent number of ballots in each ballot-box, and thus prevents any potential detection.

Second, the attacker must also make sure that the moves and replacements they may operate will not make a voting option that was intended to be expressed in ballot-box u_1 or u_2 completely disappear, as already explained in Section 8.4.2. This is addressed with Step b and Step c below. Note that we assume that at least $|\mathscr{V}|$ voters eligible in u_2 (resp. in u_1) are willing to cast a ballot, which is a reasonable assumption (see Section 8.4.3.2).

The attack [Swap$_X$] is as follows. For the target Alice eligible in u_1, the attacker first selects a ballot-box u_2, ideally with a low number of voters and does the following:

Step a: The attacker uses [Move$_X$] to move Alice's ballot b_A from u_1 to u_2 and some arbitrary Bob's ballot b_B from u_2 to u_1 (with a replacement of b_B by b_{att} if $X = B$).

Step b: For any other voter willing to cast a ballot b in u_2, the attacker exploits [Replace] to replace b by an attacker-chosen ballot b'. The attacker can choose b' such that: (i) each voting option $v \in \mathscr{V}$ gets at least one

vote in u_2 and (ii) the overall vote distribution is close to the expected one (to not raise any suspicion).

Step c (optional): If the expected vote distribution in u_1 makes it likely that a voting option $v \in \mathscr{V}$ might get no vote, then [Replace] must also be used against $|\mathscr{V}|$ voters eligible in u_1 to remedy this problem. This would not have been required for most ballot-boxes in the 2022 election (see [*DH23*, Remark 6]).

Finally, the tally of u_2 leaks Alice's intended vote.[11]

11: We additionally provide an attack variant to [Swap$_{ID}$] that does not rely on (V1) in [*DH23*, Remark 6].

8.4.3.2. Impact and Stealthiness

The impact of our privacy attacks is maximal when all plaintext votes of the ballots in the resulting ballot-box u_2 are known to the attacker, except for the one of the target voter (*e.g.,* Alice).[12] In this case, the result of the tally of u_2 directly reveals Alice's vote to the attacker. The larger is u_2, the more replacements are needed. We now discuss how suitable u_2 can be chosen using data of the election for which FLEP was used [flep22-r2] and then discuss coercion and stealthiness.

12: Note that, even if some unknown ballots remain in u_2, some privacy leak still exists. Some quantitative analyses of similar privacy leaks have been conducted [MMR23] and have revealed that it could be harmful.

Many suitable u_2. In the 2022 French legislative election, many consulates received a small number of votes, hence suitable choices for u_2. For example MSQ-MINSK during the second round (6 electronic votes among 129 eligible voters) or PBM-PARAMARIBO during the second round (5 electronic votes among 145 eligible voters). These two consulates belong to constituencies having consulates with a large number of voters with potential targets, hence suitable choices for u_1, such as SYD-SYDNEY which received 4049 electronic ballots during the second round, or MEX-MEXICO with 1959. There were even extreme cases with some consulates which received a *unique* ballot as discussed in [*DH22*, Appendix B.2]. Obviously, our attacks can be exploited against different targets voters (Alice) using different target ballot-boxes (u_2).

New coercion power. Moreover, the attacker can also target several voters instead of just Alice and gather all their ballots into u_2. This attack would give a power of coercion, in a remote way[13], against those voters: the tally of u_2 will reveal the vote distribution among them and thus make the coercer able to detect that some decided to not comply with the attacker's instructions and how many (the attacker does not learn their identity though). Similarly, the attacker can exploit this to learn the vote distribution of a population of voters of special interest, *e.g.,* industrial actors. Those votes would normally be mixed with many other votes in their respective ballot-boxes, thus protecting their privacy.

13: Note that the FLEP was not intended to guarantee coercion-resistance, which is usually understood as a resistance to over-the-shoulder coercion where the attacker is physically with the voter under coercion. However, one could have reasonably expected that it was resistant to this weak form of *remote* coercion, in which the coercer has *never* access to the voters' devices.

Why are our privacy attacks completely stealthily? We review all checks that are performed by the different actors and explain in detail why they fail to detect our privacy attacks in [*DH22*, Appendix B.2]. We summarize this study here.

From the (honest) voting server point of view, we note that data received and processed by the voting server are genuine- and honest-looking, the voting server does not know how voters intended to vote. From the voting client point of view, the received malicious ballot receipts pass all consistency checks and no error can be detected. No cheating can be detected from the ballot-boxes tallied result either since each ballot-box contains (at least) one ballot for each voting option. Therefore, even if a voter's ballot has been moved in another ballot-box or replaced, then they will always see in the result at least one vote which corresponds to their intent, it could have come from their ballot.[14]

14: The only assumption we make here is that honest voters do not collude to infer what should be the expected, honest result. Note that, the attacker can remain stealthily even under such extreme circumstances: they could rely on malicious and colluding voters in the same ballot-box to lie and cheat against the honest voters.

We now discuss public information and independent verification services proposed to the voters. The 3rd-party checks that the tally of the ballot-box has been correctly computed and fails to detect any cheating because of (V2). The voters' individual verifiability checks also fail to detect our attacks, even when carried out with the 3rd-party service. Indeed, to answer these queries, the latter recomputes all the references H and hashed values hb of the ballots provided by the authorities, which are then looked up. It is important to note that, because the 3rd-party was aiming to guarantee verifiability, this look-up is performed taking all the ballots of the given constituency identifier `elec` into account. Therefore, even malicious receipts will be found and deemed valid.

Finally, the human voters themselves are unable to detect that the ballot reference they were shown ($H_1^{s_2} = H_1^{s_4}$) are not computed with the right ballotBoxId (and the genuine, honest ballot b). Indeed, to be able to detect this, the voters would need to recompute H and thus to know what is hashed in H (*e.g.*, b) but voters are never shown b.

Impact: A channel or a voting server attacker has the technical ability to learn some target voters' vote without breaking any encryption. This attack would leave no evidence we could then exploit to know if the attack happened.

8.4.3.3. Fixing Ballot Privacy

We propose three solutions to prevent our attacks.

Fix 3: A first fix is to add the ballot-box (consulate) identifier ballotBoxId (*i.e.,* u) in the context of the ZKP. This simple modification cryptographically prevents any swap of ballots between different ballot-boxes once it has been computed.

Fix 4: Even if Fix 3 was implemented, [swapID] is still possible. Therefore, we recommend coupling Fix 3 with displaying in the voting client the ballotBoxId used for computing the ballot (or the readable name of the corresponding consulate). This way, the voter could notice the cheating prior to casting.

Fix 5: In addition to Fix 3, the 3rd-party verification service may verify the consistency between the ballotBoxId in the ZKP (and the cSU) and the ballot-box in which it is stored as introduced in the tool after the elections run in 2022 [CGGc]. Therefore, as an alternative to Fix 4, the 3rd-party could additionally display to the voter the consulate in which their ballot has been cast and counted.[15] This solution is easy to implement but only prevents privacy attacks against voters who use the 3rd-party verification service. This is of broader interest: 3rd-party, *i.e.,* an external auditor, can be useful not only to ensure verifiability, but also vote privacy.

15: Note that ballotBoxId is also in plaintext in cSU so it could be directly checked but it seems unrealistic to ask human voters to do so.

Remark 8.4.4 Fix 3 and 5 have been implemented by Voxaly Docaposte and 3rd-party thanks to our findings and were used for elections in 2023.

Fixing the vulnerability (V1) with Fix 1 or Fix 2 would already prevent all but the [swap$_{ID}$] attack. The latter requires the combination of Fix 3 and either Fix 4 or Fix 5. Moreover, we believe the second vulnerability we found (V2) should be addressed independently with Fix 3 as it introduces a weakness, even though not exploitable on its own. Therefore Fix 3 and 4 would be the best option. ▷

8.5. Recommendations and Lessons Learned

Our reverse and security analysis of the FLEP revealed serious vulnerabilities and attacks in the protocol and its deployment. This is quite impactful on its own given the importance (nation-wide legislative election) and scale (largest political election using e-voting) of this election. Our analysis of the FLEP is also relevant for its illustration of how things can go bad with the excessively challenging real-world deployment at scale of e-voting solutions. Some of them are due to pitfalls that we point out for improving future deployments of e-voting in general. Others are due to limitations and problems with state-of-the art academic e-voting solutions from which we derive practically relevant open research questions of general interest. We summarize those lessons that go beyond the specific case of the FLEP and we refer the curious reader to [*DH22*, Appendix E] for more in-depth discussions.

Voting client as critical component Great care has been put in securing the FLEP voting server against internal and external threats, which is a common practice. E-voting requires to adopt a radically different mindset in that regard as it relies on *verifiability* so that no one has to trust the voting server, its administrators, the code it runs, *etc.* (sometimes called *software independence* [Riv08]). The most important component of the protocol that should be the main target of audits and analyses is actually the *voting client* and the verification services (*e.g.,* 3rd-party).[16] If the voting client is securely designed and implemented, the protocol should guarantee the election integrity independently of the voting server security. This is well illustrated by our work and our attacks: the partial specification and 3rd-party verification services are totally voided by the voting client being flawed. An implementation weakness in the voting client can completely defeat verifiability and privacy as we have shown. *Recommendation:* Consider the voting client and the verification services as the main targets of analyses and internal audits. We advocate for making them amenable to public scrutiny with a comprehensive public specification and open-sourced code.

16: This does not dispense to make effort to secure the voting server.

The FLEP, as well as many e-voting systems assume an honest voting device (*e.g.,* [CGG19; CCM08; JCJ05]). On the surface, this seems unrealistic since the latter is distributed by a possibly compromised server or channel. Instead, it could be distributed through an app marketplace to benefit from code integrity (assuming the marketplace is trustworthy). Otherwise, when downloaded and run in the browser, it is inherently vulnerable to integrity flaws. This is resolved in the literature with the notion of "auditability" (*e.g.,* [Bel+13])[17]: *external auditors* (who can be any voter or external actors) can pretend to be voters to compare the code they receive with the legitimate version and thus implement some form of "universal integrity checks". We stress that such checks are by no means an audit of the code (that is rather the role of the commissioned auditors) but rather a mere comparison of two code bases – or the hashes thereof. If there are enough auditors well hidden among the voters, the voting server takes high risk of being caught when tampering with the voting client. In the context of the FLEP, such external audits were never specified and were made complex to carry over[18]. This can be addressed by using Single-page Application (SPA)[19] to distribute the voting client, as is the case for Belenios. Only the SPA needs to be compared to the legitimate version and can be considered as a static file, which can then be considered to be honest (assuming the legitimate version has been well audited by commissioned auditors, once-for-all). *Recommendation:* Distribute the voting client with a standalone app or an SPA and clearly

17: Note that mechanisms such as the Belanoh challenge [Ben87] do not ensure the voting client integrity.

18: The JavaScript code is served post-authentication, the voting client logic is spread all over HTML and JavaScript files, interleaved with user- and session-specific data (not easily comparable)

19: https://developer.mozilla.org/fr/docs/Glossary/SPA

specify external auditors' tasks, notably universal integrity checks.

Finally, the vulnerability (V1), which is essentially a verifiability weakness, is key to our privacy attacks. Such a verifiability-privacy relation has been documented before with a theoretical attack [CL18], our work illustrates further this relation with practical attacks. *Lesson:* Privacy can be attacked in practice due to a lack of verifiability.

Operational constraints as scientific bottleneck For deploying a protocol in the real world and especially for political elections, many aspects that are very often omitted in the academia have to be taken into account such as the legal requirements and recommendations, compliance to a competitive public call for tenders, *etc.* This partly explains why, despite being derived from the proven secure Belenios [CGG19], the FLEP ended up flawed with weaknesses that were inexistent in Belenios.

Missing features. First, the FLEP had to accommodate the multi-ballot-box setting. This setting seems to be a recurring pattern. Importantly, it is the source of quite impactful privacy attacks: [Swap$_b$,Swap$_H$,Swap$_{ID}$] for the FLEP and another privacy attack [CDG21] for the Swiss Post 2022 protocol, which also relies on multiple ballot-boxes but exploits a different weakness. Another lacking feature in Belenios was the downloadable receipt, a desired feature of the EFA Ministry from their interpretation of the CNIL recommendations. As a result, the PDF receipt has been introduced in the FLEP but was not properly checked. *Research question:* Make state-of-the-art solutions more generic so that they can accommodate such real-world use cases and practical constraints. Moreover, formal security properties and proofs do not consider such features and could thus miss practical attacks.

Distribution of authentication. The FLEP suffers from a lack of trust distribution for the voter authentication process: a single entity, the voting server, is in charge of this authentication as it stores and checks the two authentication factors. A compromised voting server can thus impersonate voters and do *ballot stuffing*. A simple theoretical solution to fix this weakness is to distribute the generation and the verification of the authentication factors. Unfortunately, deploying such an infrastructure is difficult. For example, if the independent 3rd-party entity had to participate to user authentication, then it would have to offer an API for authentication verification. As far as we know, there was no practical academic solution to this problem that matches real-world constraints of deployments at the time we published [*DH23*]. We envision the spread of eID cards or the development of standards like openID connect [Sak+14] could be built upon for this. We have later proposed an eligibility verifiability mechanism based on openID connect in [*Cor+24*], see a brief summary in Section 8.6 that contributes to addressing this problem.

Secret Key Generation distribution. We show in [*DH22*, E.3] that some lawful requirements governing when a quorum is met to *e.g.*, decrypt a ballot-box are not cryptographically enforced in the FLEP. (The lawful requirements combine conditions such as: a holder and a substitute for a given role cannot be simultaneously in the same quorum, a quorum should have at least 4 roles represented, *etc.*) As a consequence, fewer people than what is legally prescribed by the law can collude and decrypt a ballot-box. State-of-the-art academic solutions cannot be used off-the-shelf to address this. *Open Questions:* Is it possible to cryptographically enforce such operational constraints to prevent any human misbehavior? Are there solutions that are generic enough to be suitable for practically relevant constraints seen in real-world use cases? We formalize such questions in [*DH22*, Appendix E.2.1].

Clear security objectives and threat model Neither the FLEP nor legal requirements and recommendations clearly define the expected security objectives and threat models. It is meaningless to assess the security of a system without a clear threat model definition, that is why we first had to do it ourselves. It is also of little interest for the public to know that the FLEP has undergone audits without having access to the scope and the objectives of this audit. *Recommendation:* Election organizers, or even better the responsible for the public call for tenders (EFA Ministry), should clarify and specify their expectations regarding the security objectives and threat model. This will certainly help academic researchers to identify and address practically relevant problems, incentivize more secure solutions in calls for tenders, and allow more relevant audits and security analyses. Ideally, objectives and threat models could be prescribed by law, as it is the case in Switzerland (see [*DH22*, Remark 14]), where even mathematical cryptographic and symbolic proofs are required. We also recommend requiring such proofs.

Simpler and precise voting journey instructions In the FLEP, the voters receive different confusing and quite overwhelming information about the verification process. In total, they are shown 4 quite intimidating cryptographic data and are offered 4 different verification tasks, see [*DH22*, Appendix E.3.1]. *Recommendation:* Complex protocol and notably voter's journey and verification tasks are detrimental to the overall protocol security. For the FLEP, we explain in [*DH22*, Appendix E.3.1], how the protocol can be conservatively simplified with only one cryptographic datum shown to the voters and a unique verification task, which would certainly have avoided the vulnerability (V1). In general, we recommend prioritizing any simplification impacting the voters' journey and clearly specifying what is expected from the voters and assume no more from them.

Transparency and openness The FLEP was lacking a clear (and open) specification, which partially explains why its flaws remained unnoticed until we saw them, while the election was running. (We present in [*DH22*, Appendix D] additional weaknesses that are well-known in the literature, *e.g.*, ballot replay attacks, and that nonetheless also affect the FLEP.) *Recommendation:* Because designing and deploying e-voting solutions is a notoriously difficult task, we recommend promoting transparency and public scrutiny from different communities (academic researchers, hackers, *etc.*) to detect and prevent vulnerabilities as early as possible. This could be incentivized with public intrusion test and bug bounty programs, as done in Switzerland where open access is even a legal requirement.

8.6. Breaking and Fixing Other E-Voting Components

Improving Eligibility Verifiability with OpenID Connect [*Cor+24*] One of the open question we raised in Section 8.5 was to design practical means to guarantee eligibility verifiability, that is ensuring that all recorded ballots were cast by distinct eligible voters. Indeed, current solutions are either impractical or rely on trusting the voting server, in which case the voting server can easily stuff the ballot box. This is still the case in deployed protocols such as the FLEP. We then proposed [*Cor+24*] a new cryptographic sub-protocol to achieve this property by building on an existing authentication infrastructure (OpenID Connect). We provide a full-fledged proof of concept, evaluation, and a formal security analysis performed with ProVerif.

Attacking and Fixing Public Bulletin Boards (BBs) [*HSB21*] Another crucial aspect of e-voting protocols is the BB, which an append-only log of all cast ballots allowing the election results to be verifiable. In the case of FLEP, the BB is not made public but only accessible to the third party auditors, which are then untrusted to perform the verifiability checks correctly and to enable voters to perform their own verifiability checks towards the auditors. However, for many real-world voting protocols such as Helios or Belenios, the BB is made public for all to read and perform verifiability checks. We have shown [*HSB21*] that the such public BB in e-voting has been overlooked or treated far too casually in previous designs and analyses.

First, we have demonstrated that there is a mismatch between BB assumptions in verifiability claims and proofs and actual BB realizations. By considering an idealized BB such as a shared, universally, append-only accessible memory or a broadcast channel, prior works fall short of capturing realistic use cases and deployments. Idealized BBs are fine in theory but not in practice. In particular, a malicious BB can provide, independently of its current state, different readers different contents; we call this *equivocation*, inspired by the terminology used for distributed systems [Chu+07; RKS15]. This opens even well-designed protocols to serious attacks. In particular, we present novel and practical attacks on the state-of-the-art e-voting systems Belenios [CGG19], Civitas [CCM08], and Helios [Adi08]. In our attacks, the BB equivocates contents, but never the *election result* (*i.e.,* number of votes for each candidate/choice) to readers. We thereby show that these systems fail to provide verifiability (and privacy) under the threat models for which they are claimed to be secure. As verifiability has previously only been proven with some idealized BB, these attacks were missed in prior formal analyses.

To enable the analysis of protocols under reasonable assumptions about the BB, we introduce a new security definition that is applicable to arbitrary BBs. We identify a new requirement for the BB and formally prove that it is sufficient and, in most cases, necessary to achieve security. We then propose a BB protocol that satisfies this requirement under weak, realistic trust assumptions and provide a machine-checked proof thereof. Our protocol can be used as a replacement for existing BBs, which would secure elections under much weaker trust assumptions.

8.7. Responsible Disclosure and Further Impact

All the vulnerabilities we found in the FLEP have been reported after the election and at least 3 months before publication to the relevant stakeholders, namely EFA Ministry, ANSSI, Voxaly Docaposte, and the researchers running the 3rd-party services (Stéphane Glondu, Pierrick Gaudry, and Véronique Cortier).

May 21, 2022: Publication of Partial Specifications. The vendor (Voxaly Docaposte) released a partial specification document [Doc22]. This document was intended to facilitate the development of the universal verifiability tool by the third party and served as a reference point for our subsequent security analysis.

May 23 – 27, 2022: 2022 Legislative Elections, Data Acquisition, Starting Reverse Engineering. With the assistance of volunteer voters residing abroad, we collected network traffic logs and client-side JavaScript code during the two election rounds of the online

vote, as well as during the testing phase prior to the election. This data allowed us to analyze the actual implementation of the protocol, compare it against the partial specifications, cross-compare the different versions (test version with deployed version) and reverse the protocol. We also start our security analysis.

July 11, 2022: Initial Disclosure and Coordination. We submitted a confidential report to CERT-FR (ANSSI), invoking the protection of the Article 47 of the French Law for a Digital Republic, detailing the vulnerabilities we found. ANSSI acknowledged and established a coordination channel with EFA Ministry and the impacted vendor Voxaly Docaposte's CEO.

August – September 2022: Technical Meetings and Clarifications. Two technical in-person meetings were held involving all of these stakeholders and us to discuss the findings and potential remediations. All stakeholders acknowledged the validity of our analysis and of the vulnerabilities. Following these exchanges, we submitted a detailed document to the EFA Ministry to confirm our understanding of the system's specifications and to suggest our fixes and countermeasures.

September – November 2022: Publication Agreement and Draft Review. We coordinated with the EFA Ministry to obtain specific authorization to publish snippets of the voting client's JavaScript code. In late 2022, we shared our draft academic paper with all stakeholders to ensure the factual accuracy of the system description and the status of corrections before making our preprint public and before submitting our work to the Real World Crypto 2023 conference and then to USENIX Security 2023, where it was accepted for both conferences. We publish our first preprint of this work on November 28, 2022 [*DH22*].

March 2023: 2023 Partial Legislative Elections, Protocol Update, and Fixes. Upon the release of the updated system specification (v2.04) [Doc23] for partial elections (for the 2022 elections that were contested), we confirmed to ANSSI that the critical Swap attacks had been effectively mitigated. However, we highlighted that the implemented fixes for individual verifiability remained insufficient and that voter instructions could be misleading; in short the Replace and Drop attacks were still feasible under certain conditions. In short, these attacks are only partially mitigated in this and future versions of the protocol (see Remark 8.4.3).

Late March 2023: RWC 2023. Presentation of our work at the Real World Crypto Symposium 2023.

August 2023: USENIX Security publication. Our paper is presented at and published in the proceedings of the USENIX Security 2023 conference [*DH23*].

May 22, 2024 CNIL-Inria Prize. Our work was formally recognized by the CNIL and Inria. We received the **CNIL-Inria Data Protection 2023 Award** at the 17th Computers, Privacy and Data Protection (CPDP) conference for our paper on FLEP [*DH23*]. The jury emphasized the very tangible impact of this work on the integrity and confidentiality of votes, and also highlighted our constructive collaboration with stakeholders.

June – July 2024: 2024 Legislative Elections, Operational Monitoring, New Issues. Following the dissolution of the National Assembly, the system was used again in June 2024 and was based on the 2023 version of the protocol. The deployment happened in a very short time frame. We have reported two anomalies regarding the server seal verification to the stakeholders.

We first alerted about the unavailability of verification servers immediately following the vote counting ceremony, which made it impossible for voters to perform their individual verifiability after the election, contrary to CNIL recommendations. This issue is worsened by the fact that no third-party auditor was appointed to run independent verification servers for the 2024 election.
We also alerted ANSSI and EFA Ministry regarding an inconsistency in the verifiability check web-service: a voting receipt from 2022 (sent for testing purposes) was cryptographically validated by the server, despite the ballot obviously being absent from the 2024 ballot box. This hints at a way too lax verification logic on the server side, which is likely using the public key provided within the seal itself[20], making the security claim misleading for voters. Even assuming an honest voting server, this issue undermines the trustworthiness of the individual verifiability mechanism since voters obtaining fake receipts from fake voting servers (hence signed with a fake key) would be deemed valid by the official and honest verification service. The issue was immediately reproduced and acknowledged by ANSSI. However, due to the short time frame before the election (less than 2 weeks), no fix could be implemented for the 2024 election.

We thank those stakeholders, *i.e.,* EFA Ministry, ANSSI, Voxaly Docaposte, and the researchers running the 3rd-party services (Stéphane Glondu, Pierrick Gaudry, and Véronique Cortier) for their help and discussions after we sent them our findings. In particular, we would like to thank again the role of ANSSI in the responsible disclosure process, which has always been a key player in promoting transparency and openness. This is greatly appreciated given the context of this work.

Later Collaboration with ANSSI Since we responsibly disclosed our attacks and fixes, we closely work with ANSSI about other key aspects of e-voting security (*e.g.,* "vote-tagging"[21], eligibility verifiability, *etc.*). We also work as an academic consultant to help ANSSI define its latest technical doctrine[22] on the security of e-voting protocols. We also have contributed [*Cor+25*] to the call to participation by the CNIL about its latest recommendations[23].

Ethical Considerations To validate the findings without interacting with the live election servers (which would violate the "good faith" requirement of Article 47 of the French Law for a Digital Republic), we limited ourselves to a passive analysis only; we never attacked voting servers. We then explicitly invoked the protections of Article 47 of the French Law for a Digital Republic by choosing to report directly to ANSSI rather than the vendor or the press, to ensure legal immunity and national security coordination. We have followed all best practices for responsible disclosure, including non-disclosure until fixes were implemented and agreed upon with the stakeholders, as detailed above.

Media Coverage Our work [*DH23*] got some press coverage: Inria frontpage, l'Est Républicain (here and there, in French), and a podcast (in French). We also wrote a popularization article about this and e-voting in general in the online newspaper The Conversation (in French).

20: A colleague of ours affectionately nicknamed this the "Noddy server" (after the "Noddy" character, *Oui-Oui* in French), as it naively nods "yes" to every receipt it receives, regardless of its origin—providing a false sense of security rather than any actual guarantee.



21: When ballots are tagged with voter-attributes such as gender or location in order to produce demographic statistics or produce sub-results.

22: This document will be finalized and made public by the end of April. In the meantime, a preliminary version is available here.

23: The finalized document is also expected to be published by the end of April, see this "délibération".

8.8. Conclusion

The FLEP protocol is an instructive example of a real-world deployment of a variant of an academic protocol (Belenios) that introduces design-level and implementation-level weaknesses. The latter opened up the 1.6 million eligible voters overseas of the 2022 French legislative election to integrity and privacy attacks under a voting server attacker or a weaker channel attacker. To avoid such failures in the future, we proposed fixes, which have been (almost fully) implemented, and have made various recommendations we learned from this experience. We had insightful discussions with the different stakeholders and we strongly believe more of such communication between academia and e-voting practitioners is for the better.

However, we must remain cautious as, in the absence of formal security proofs, it is still possible that other critical attacks might affect the current version of the FLEP protocol. Narrowing down to the weaknesses we know of, we recall that better solutions should be proposed for authenticating the voters, getting rid of the strong assumption of a trustworthy voting client, *etc.* Indeed, in its current state, the FLEP protocol still allows a compromised voting server to stuff the ballot-box by impersonating voters who do not vote. Such an attacker can also modify the voters' intended vote having a corrupted voting device, *e.g.,* because of a malicious web browser extension. Moreover, unlike the flaws we discovered, we consider that there is currently no off-the-shelf solution to prevent such attacks. This might be considered as a warning that the state of the art in e-voting is not ready for such critical elections and an incentive to continue the research towards practical solutions to those problems.

From an academic perspective, we believe that this analysis highlights the importance of bridging the gap between academic research and real-world deployments. It has notably highlighted new research questions that we believe are important to address to improve the security of e-voting systems in practice. For instance: How to handle the multi-ballot-boxes and multi-sub-elections setting without compromising on ballot privacy and verifiability? How to deal with *vote-tagging* with similar security? How to cryptographically enforce some of the lawful operational constraints (*e.g.,* strict rules defining quorums) which are currently only enforced procedurally? How to formally model and verify e-voting protocols with such complex operational constraints and mechanisms?

Part C.

Formal Models Meet Fuzz Testing

Contents of this Part

9 Dolev-Yao Fuzzing 154

9.1 Background 154

9.2 Dolev-Yao Model Abstraction Amenable to Fuzzing 157

9.3 DY Fuzzing Design 161

9.4 Brief Overview of Differential DY Fuzzing 167

10 Implementation, Evaluation, and Case Studies 171

10.1 The PUFFIN DYF Library and its Fuzzers 171

10.2 Evaluation of DY Fuzzing 174

10.3 Evaluation of Differential DY Fuzzing 180

10.4 Summary of Findings and Responsible Disclosure 183

11 General Perspectives 185

11.1 Formal Verification Perspectives 187

11.2 E-Voting Protocol Perspectives 190

11.3 Dolev-Yao (DY) Fuzzing Perspectives 191

11.4 Bridging the Gap Between DY Fuzzing and Verification . . . 196



Figure 8.3.: PUFFIN Fuzzer: Finding all your fishy code. "Puffin headshot" by Steve Garvie, CC BY-NC-SA 2.0.

Cryptographic protocols are extremely hard to get right and this applies to their design, as we have learned over the last chapters, but also to their implementation. While the previous chapters focused on design-level flaws in cryptographic protocols, and formal verification techniques to preclude them, we now turn to implementation-level flaws. Indeed, specifications are simply an abstraction of the program that end-users deploy and run, programs that are themselves plagued with frequent implementation bugs. As mentioned in Section 1.1, these bugs introduce additional vulnerabilities, notably ***implementation-level logical attacks***, that are attacks captured by a DY attacker (*i.e.,* network attacker treating exchanged packets as structured messages) exploiting implementation bugs. Examples thereof are abundant and well illustrated by the extensive history of such attacks on Transport Layer Security (TLS) implementations that exploit implementation-level protocol flaws, [CVE14-1266; Beu+15; RP15; Fit+20; CVE09-3555; MDK14; BSY18; RLD22].[24] The French Legislative E-Voting Protocol (FLEP) attacks presented in Chapter 8 are also interesting examples, notably the (V1) vulnerability enabling the verifiability attacks (A1) and (A2), which is due to an implementation flaw in voting client software (see Section 8.4).

More generally, even a formally proven specification can result in a flawed and insecure implementation. Sadly, fundamental problems in program verification are still unsolved and coping with (combinations of) features of real-world systems such as pointer aliasing or complex control-flow is still out of scope, precluding verification of most existing and *currently deployed*, real-world code bases such as OpenSSL, BoringSSL, WolfSSL, and LibreSSL we aim to test.[25] For such code bases, those techniques are thus limited to the design level but not the implementations thereof; we expand on this in Section 9.1.3 when we discuss related works.

Memory-related Vulnerabilities and Fuzzing In this Part, we are concerned with finding *implementation-level DY attacks* in large cryptographic protocol code bases. To attack this problem, we draw on ***fuzz testing*** [Fio+20; Zal16; LLV; Man+21], a family of techniques that emerged in the early 1990s [MFS90] and has since established itself as the de facto standard for stress-testing security-critical software. Fuzzing is now embedded in mainstream software-assurance practice: major vendors such

24: For instance, Apple's "Goto Fail" attack (CVE-2014-1266) originated from a duplicated `goto fail;` statement in `SSLVerifySignedServerKeyExchange` that caused the function to short-circuit and return success without ever checking the server's signature on the Server Key Exchange message. The flaw silently disabled authentication along the pre-TLS 1.2 ephemeral Diffie–Hellman handshake path, enabling a network attacker to mount a Machine-in-the-Middle (MIM) attack by injecting a forged Server Key Exchange with attacker-controlled Diffie-Hellman (DH) parameters and impersonating any server to a vulnerable client. Detecting it requires reaching that specific code path with a carefully crafted sequence of handshake messages (notably negotiating an ephemeral DH cipher suite from a pre-1.2 TLS version) and identifying a semantic authentication bypass, rather than merely observing crashes or memory corruption.

25: These are very large, industrial-strength software projects with hundreds of thousands to more than a million lines of code. They are primarily implemented in low-level systems languages (mostly C, with some C++ components), which further increases verification complexity in practice. At this scale, fully verifying the deployed implementations end-to-end is still beyond what current methods can realistically handle.

as Google [Bab+19], Adobe, Cisco, and Microsoft run it continuously and at scale, both on their own products and on third-party dependencies. Its appeal stems from a combination of scalability and an unmatched effectiveness at uncovering *spatial and temporal memory-safety bugs* — a class of defects that is pervasive in modern software yet notoriously hard to surface through manual review. Concretely, most fuzzers are structured around a *fuzzing loop*: inputs drawn from a *Corpus* are mutated at random and replayed against the Program Under Test (PUT).[26] A *feedback* signal —typically code coverage— decides whether a mutated input is interesting enough to be promoted back into the *Corpus*, while an *objective oracle* monitors each execution for violations of a fixed *security policy* (most commonly, memory corruption).

26: We use *bit-level fuzzers* to refer to fuzzers whose test cases are raw bitstrings and whose mutations are atomic bit-level operations, such as bit flips or byte substitutions.

However, state-of-the-art fuzzers are unable to capture the class of ***implementation-level DY attacks*** for two main reasons. First, they do not faithfully capture the DY attacker, in particular its ability to perform structural modifications on messages for instance through the use of cryptographic primitives (*e.g.,* re-signing a message with an adversarially controlled key), which is required to realize DY attacks. logical attacks may trigger memory-corruption vulnerabilities, but they are not limited to them; they may also induce protocol vulnerabilities that do not involve memory errors. Even setting aside this first reachability limitation, state-of-the-art fuzzers are typically designed to detect the former, but generally fail to detect the latter, *i.e.,* protocol-level security violations such as authentication bypasses. (We discuss this gap further in Section 9.1.3.)

A new Fuzzing Approach: DY Fuzzing and PUFFIN

We have answered the lack of effective techniques to preclude logical attacks from cryptographic protocol implementations with the following contributions that were first published in [*AHK24*] and then extended with differential fuzzing [*GHK27*] and our PUFFIN [puffin] software.

[*AHK24*]: Ammann, Hirschi, and Kremer, *DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing*, S&P'24

[*GHK27*]: Gouville, Hirschi, and Kremer, *DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols*, S&P'27

[puffin]: puffin team, *TLSPUFFIN: a Dolev-Yao model-guided fuzzer for cryptographic protocol implementations:* https://github.com/tlspuffin/tlspuffin

DY Fuzzing Design We propose a novel approach to fuzzing cryptographic protocols dubbed Dolev-Yao model-guided Fuzzing (DYF) (or *DY fuzzing*). It is based on the novel idea of using formal DY models as domain-specific knowledge to guide the fuzzer and give it the ability to detect logical attacks in protocol implementations. This requires a redesign of most of the fuzzer components.

The **test cases** become formal and structured *DY traces*. Those represent executions of protocols against a DY attacker in the (formal) DY model: networking actions (inputs, outputs) and adversarial manipulations over exchanged messages that are abstracted away by terms and black-boxed cryptography (see Section 2.3). The DY fuzzer is thus capable of intercepting, eavesdropping, modifying, and synthesizing messages, treating them as formal terms, and of actively participating in protocol sessions. We design new domain-specific **mutations** over DY traces for exploring the search space made of all the DY traces. Any DY trace can be compiled into a proper *test case* against the PUT through **concretization**: DY terms are evaluated into bitstrings, and DY inputs and outputs are evaluated into communications with agent instances of the PUT.

The *messages as terms* paradigm guides DY attackers and thus our DY fuzzer in how they interact with messages and cryptography. It is on purpose that our fuzzer inherits this standard DY assumption: it narrows down the search space to DY attacks and excludes some other attacks, *e.g.,* when bit-level transformations are required and cannot be described at

the term-level. The DY attacker model and the corresponding attacks are grounded in decades of research and practical attack discovery [Bar+21a], including our own attacks discussed in previous parts, and supported by evidence from the broader literature and real-world protocol analyses.

Next, we define domain-specific **fuzzing security policies and objective oracles**. Indeed, the usual policies and objective oracle for spatial and temporal memory errors (*e.g.*, AddressSanitizer (ASAN) [Ser+12]) are unable to detect *logical attacks triggering protocol vulnerabilities*. We still enable them, as they are useful to find *logical attacks triggering memory vulnerabilities*, *e.g.*, memory bugs that can only be triggered from specific states that are hardly reachable using standard fuzzing. As we shall see, DY traces are very good at exploring such *deep states*. To be able to detect protocol vulnerabilities, we additionally consider formal DY properties, *e.g.*, *agreement*, that express the absence of protocol vulnerabilities. Those properties are translated into security policies and objective oracles. This way, any violation thereof reached through fuzzing is detected and flagged as an attack candidate. We stress that a DY fuzzer does not require a complete protocol DY model, but only a DY attacker model (by means of a term algebra and agents the attacker can communicate with) and security policies.

PUFFIN: A Full-fledged and Modular DY Fuzzer Library and Implementation In addition to a generic design specification for DY fuzzers, we also contribute a complete Rust implementation (PUFFIN) of a DY fuzzing engine that can be specialized to different protocols and PUTs. We have notably specialized PUFFIN to TLS, yielding TLSPUFFIN that we applied on four different PUTs: OpenSSL, BoringSSL, WolfSSL, and LibreSSL. We have also built a preliminary SSH PUFFIN for the SSH protocol (with LibSSH and WolfSSH as PUTs) and Open Platform Communication Unified Architecture (OPC UA) PUFFIN as a first proof of concept for the OPC UA protocol (with Open62541 as a PUT). This whole FLOSS project [puffin] is ca. 60k LoC, 48k of them in Rust (see Table 10.1). Our implementation follows modular design principles and revolves around three main modules that are of independent interest.

[puffin]: puffin team, *TLSPUFFIN: a Dolev-Yao model-guided fuzzer for cryptographic protocol implementations:* https://github.com/tlspuffin/tlspuffin

First, the protocol- and PUT-agnostic DY fuzzer module PUFFIN is built on top of the fuzzing library LibAFL [Fio+22], where we re-implemented most of the fuzzing components. The implementation of PUFFIN (ca. 17.2k LoC) is written once and for all and can be reused for any protocol and PUT; for this, we designed minimal interfaces to plug in protocol-specific components.

Second, we built on top of PUFFIN protocol-dependent fuzzers: TLSPUFFIN for TLS 1.2 and TLS 1.3, SSHPUFFIN for SSH, and a preliminary Proof-of-Concept (PoC) OPCUAPUFFIN for OPC UA. Specializing PUFFIN to new protocols requires protocol-specific components such as security policies and a concretizable term algebra, which we generally obtain by wrapping existing cryptographic libraries. The required effort varies by protocol and amounts to thousands of Rust LoC (ca. 6.5k for TLS, ca. 13k counting boilerplate code and tests), but this work is then reusable for any PUT of the same protocol.

Third, PUFFIN offers an interface for *harnessing* a PUT, which is the process of connecting the fuzzer to the PUT and making it execute test cases against it. The effort is rather low (for TLS, ca. 1.5k LoC per PUT on average) and is mostly reusable for any version of the same PUT. We connected several PUTs to our fuzzers this way: OpenSSL, BoringSSL, WolfSSL, and LibreSSL to TLSPUFFIN; LibSSH and WolfSSH to SSHPUFFIN.

Our fuzzer is fast—hundreds of executions per second per CPU core—and allows parallel processing. Beyond fuzzing, it offers additional features for *e.g.,* executing custom traces against a PUT or even through a TCP connection, graphically visualizing test cases, and exporting bitstrings of messages to help build PoCs.

We extensively evaluated our fuzzer, in particular TLSPUFFIN on TLS 1.2 and TLS 1.3 implementations. We let TLSPUFFIN run on the aforementioned TLS PUTs, and found 11 vulnerabilities, including 6 new CVEs on WOLFSSL (among them one critical and two high-severity). We also ran three state-of-the-art stateful protocol fuzzers[27] and one test-suite and confirmed that none was able to find any of these vulnerabilities. We explain why they were outside their scope due to inherent limitations of previous approaches. We perform benchmarks comparing our fuzzers to other stateful protocol fuzzers. TLSPUFFIN achieves a similar amount of code coverage, but the covered code is often incomparable. However, our analysis shows that code coverage is an unsatisfactory metric to evaluate the DY fuzzer's ability to find attacks (which was already noted for standard, bit-level fuzzers [Kle+18]).

27: These are fuzzers designed for network protocols. We discuss the most relevant ones in Section 9.1.3.2.

Differential DY Fuzzing A limitation of protocol-dependent objective oracle checking for DY security policies is the requirement to precisely model properties to check at runtime (*e.g.,* agreement on ession parameters). Defining these properties is labor-intensive and inherently non-exhaustive. Consequently, typically only a subset of logical attacks is detected in practice, as was the case for our first DYF design [*AHK24*].

We address this limitation by introducing Differential Dolev-Yao Fuzzing (DDYF) based on a differential oracle to compare executions across different protocol implementations [*GHK27*]. By interpreting discrepancies through the DY model, it identifies semantic behavioral differences indicative of bugs or vulnerabilities, effectively minimizing false positives.

[*GHK27*]: Gouville, Hirschi, and Kremer, *DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols*, S&P'27

We propose a generic design for DDYF, implement it within the PUFFIN DY fuzzer, and evaluate it on four major TLS and two SSH implementations mentioned above. Our results demonstrate that DDYF can detect vulnerabilities that evade state-of-the-art fuzzers, including DYF, specifically those requiring DY attacker capabilities (missed by bit-level fuzzers, even with differential oracles) or complex objective oracles (missed by non-differential DY fuzzing). DDYF also uncovered new security-related bugs that prompted upstream fixes by the vendors upon our responsible disclosure. For instance, DDYF flagged behavioral differences that turned out to reflect four distinct postures toward the client-side defense (RFC 5746) against the CVE-2009-3555 renegotiation attack: OPENSSL enforces it by default (since 2021), LIBRESSL disables it through an option that is set by default, WOLFSSL compiles it only under an opt-in hardening flag, and BORINGSSL hardwires it off entirely. Next, DDYF found that LIBRESSL's Elliptic Curve implementation lacks a second line of defense, computing with the private key on an adversarial degenerate point before rejecting it. After our reports, both were addressed upstream by all affected vendors. Finally, DDYF discovered 52 new acknowledged RFC violations in TLS and SSH stacks (47 and 5, respectively). Note that specification compliance cannot be checked in general by standard *Objective Oracles* while DDYF turns out to be particularly well suited for finding such bugs.

Contributions To summarize, our contributions are as follows.

1. We propose DY Fuzzing: a new approach to fuzzing cryptographic

protocols that notably captures for the first time a DY attacker and the class of DY attacks. We propose a new and complete system design.
2. We propose PUFFIN and TLSPUFFIN: a full-fledged, modular, and efficient implementation in RUST of this fuzzer design.
3. We evaluate and compare TLSPUFFIN on several TLS 1.2 and TLS 1.3 libraries and (re)found 11 vulnerabilities not found by others, including six new ones (one critical, two high, and one medium).
4. We design Differential DY Fuzzing (DDYF) that extends DYF with a differential *Objective Oracle* and implement it in PUFFIN. DDYF (re)found vulnerabilities missed by state-of-the-art fuzzers and uncovered 52 new RFC violations, including four security-related bugs.

PUFFIN and its family of fuzzers are the fruit of a sustained collaborative effort. The project was launched under the PROTOFUZZ ANR JCJC (PI: L. Hirschi, 2023–2027), with foundational contributions from Max Ammann (Master's internship at the time, then co-author of [*AHK24*]), and has since grown through our contributions and the work of PhD students (Tom Gouville, Vincent Diemunsch), research engineers (Michael Mera, Olivier Demengeon, and Nataël Baffou), and several Master/Bachelor interns, supervised by us and S. Kremer. Building on this foundation, PUFFIN is now under active development as a long-term research platform, with ongoing work on new protocols, new PUTs, and methodological extensions of DY fuzzing (see Chapter 11). The project website[28], together with the user and developer documentation[29], are maintained in sync with the codebase[30] and serve as the canonical entry points for users, contributors, and future students joining the effort; the website also acknowledges all active and past contributors to the project.

28: https://tlspuffin.github.io/

29: https://tlspuffin.github.io/docs/overview

30: https://github.com/tlspuffin

Outline Chapter 9 is dedicated to the design of DY fuzzing: we first recall some background about TLS and fuzzing and discuss related work and then propose a generic design for DYF and DDYF. Chapter 10 discusses our implementation of DYF and DDYF in the PUFFIN library and family of fuzzers and then presents our qualitative and quantitative evaluation and comparison with state-of-the-art protocol fuzzers.

9. Dolev-Yao Fuzzing

In this Chapter, we briefly review the background on the TLS protocol, fuzzing, and related work (Section 9.1). We then present, in Section 9.2, a generalization of the DY model that forms the basis of the DYF design introduced in Section 9.3.

9.1 Background 154
9.1.1 Case Study: The TLS Protocol 154
9.1.2 Fuzzing 155
9.1.3 Related Work 156
9.2 Dolev-Yao Model Abstraction Amenable to Fuzzing 157
9.2.1 Term Algebra 157
9.2.2 DY Traces 158
9.2.3 DY Security Properties . . 161
9.3 DY Fuzzing Design . . 161
9.3.1 Mapper 162
9.3.2 Harness 164
9.3.3 DY Traces and Seed Corpus 164
9.3.4 Mutations 165
9.3.5 Security Policies and Objective Oracle 167
9.3.6 Big Picture: The DY Fuzzing Loop 167
9.4 Brief Overview of Differential DY Fuzzing . 167
9.4.1 Architecture and Differential Oracle 168
9.4.2 Key Challenges and Solutions 169

9.1. Background

9.1.1. Case Study: The TLS Protocol

TLS [Res18] is a protocol to establish a secure channel between two agents, a client and a server, communicating over an untrusted network. It is notably widely used in the context of HTTPS and enables browsers to securely communicate with web servers. TLS aims at providing strong security guarantees. *(i) Authentication*: the server is always authenticated, and the client can be optionally authenticated. Authentication is realized by the means of asymmetric cryptography (*e.g.,* RSA) or a symmetric Pre-Shared Key (PSK). *(ii)* Integrity and confidentiality: application data is always encrypted and integrity-protected with a session key. For simplicity and conciseness, we focus on the latest TLS 1.3 version.

The TLS protocol has two sub-protocols: first, the *handshake* protocol negotiates cipher suites, authenticates the end-points, and establishes a shared, session key; then, the *record layer* protocol uses the established secure channel (based on the session key) to exchange application data. With the aim of being agile and adaptive to the use case, the protocol offers different modes of operation that sometimes can be combined. This yields a rather complex state machine for clients and especially servers [Res18, Appendix A]. We give a bit more details about the handshake protocol as we will use it for illustration throughout the Chapter.

Key Exchange and Authentication The handshake protocol starts with two *key exchange* messages. The client sends a ClientHello message containing proposed cipher suites, and either an ephemeral (EC)DH key share or a PSK (or both). The server replies with a ServerHello indicating the negotiated connection parameters and, if needed, its ephemeral (EC)DH key share. Alternatively, if the client's proposal does not suit the server (*e.g.,* unsupported cipher suites) the server may send a HelloRetryRequest. Any messages after a successful key exchange will be encrypted.

To avoid a MIM attack, if the key was not pre-shared, the server must authenticate. For this it sends a Certificate, *e.g.,* a X.509 certificate, and a CertificateVerify, that is a signature of the entire handshake with the certified key. (Optionally the server may request similar messages from the client.)

Finally, both the server and client send a Finished message. This message is a MAC over the entire handshake that provides key confirmation and also binds the participants' identities to the session key.

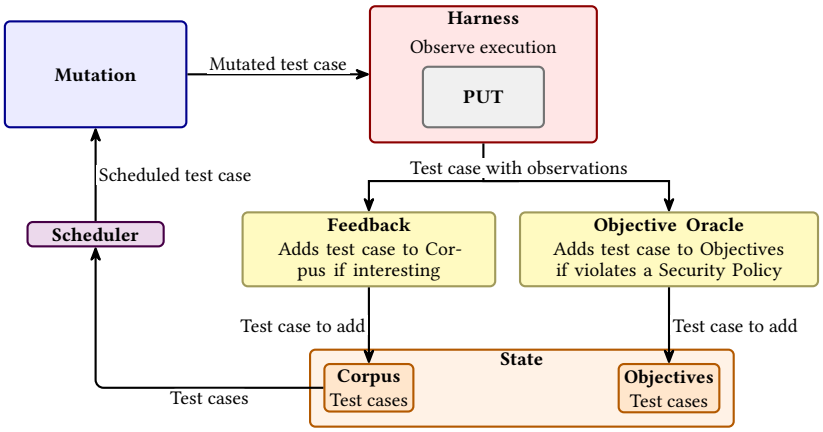


Figure 9.1.: Generic fuzzer architecture (terminology based on LibAFL [Fio+22]). The *Scheduler* picks a test case from the *Corpus* and the *Mutation* stage produces a mutated variant, which the *Harness* executes on the PUT while observing its execution. The resulting observations are analyzed by two components: the *Feedback*, which adds the test case to the *Corpus* when deemed interesting (*e.g.,* new code coverage), and the *Objective Oracle*, which stores the test case in the *Objectives* when it violates a *Security Policy* (*e.g.,* a crash or a memory error).

Session resumption At the end of the handshake the server transmits a NewSessionTicket. This ticket may either contain a key identifier, or an encrypted key for the server to allow stateless resumption. This allows for the more efficient PSK mode in a subsequent session as it avoids the certificate-based authentication.

TLS Implementations The most widespread implementation of TLS is OpenSSL with an almost 25-year history. LibreSSL is a fork thereof and aims to be more secure but has less features. boringSSL is Google's fork of OpenSSL, focusing on security and simplicity for modern use cases, and is widely deployed in Chrome and Android. wolfSSL is a lightweight implementation widely used by IoT and embedded devices, and is able to run on OSs and CPUs otherwise not supported.

9.1.2. Fuzzing

Concretely, a *fuzzer* drives the PUT on inputs that lie *outside* its honest, expected input space, looking for executions that violate predetermined *security policies*. Modern fuzzers typically proceed iteratively: at each step, fresh test cases are derived from the current *Corpus* —a curated set of "interesting" inputs accumulated so far— and run against the PUT, whose execution yields *Feedback* that, in turn, informs the generation of subsequent test cases. The signal carried by this feedback can take many forms, ranging from coverage measurements (notably *code-coverage*) to direct policy violations.

The literature distinguishes two broad families of input-generation strategies. *Generation-based fuzzers* synthesize inputs from an explicit specification of the input space (*e.g.,* a grammar), whereas *mutation-based fuzzers* apply (typically random) transformations to existing test cases drawn from the *Corpus*, promoting any mutant that the feedback flags as interesting back into the corpus. Finally, an *Objective Oracle* checks *Security Policies* when executing the test cases to detect security violations in the PUT, such as a buffer overflow.

We are mostly using the terminology of LibAFL [Fio+22], which is a state-of-the-art, modular, efficient fuzzing framework we build on. In its

default configuration, LibAFL implements *evolutionary fuzzing*, which is mutation-based and grey-box, *i.e.,* it uses some limited PUT runtime information to collect feedback (*e.g.,* through code instrumentation). The main LibAFL fuzzing loop is depicted in Figure 9.1.

9.1.3. Related Work

9.1.3.1. Fuzzing Protocols at the Bit-level

Narrowing down the discussion to cryptographic protocols, the main approach is mutation-based fuzzing on the network packets or the inputs of cryptographic primitives [PBR20; Vra20; boof; fuzzw]. Such testing methodologies are adequate to find safety vulnerabilities with potential security implications (*e.g.,* HeartBleed [CVE14-1266] and CloudBleed [CVE14-0160]), but are unable to capture logical attacks for two fundamental and intrinsic reasons given next, which we will revisit and exemplify in Section 10.2.1.5. (We provide a more thorough qualitative and quantitative comparison with state-of-the-art protocol fuzzers in Chapter 10.)

(1) Reachability Problem: logical attack states (triggering memory or protocol vulnerabilities) are **not reached** for two main reasons. *(1a) Existing harnesses* of many state-of-the-art bit-level fuzzers [Fio+20; Zal16; LLV; Cha+17; Yah+21; Asc+19] only send one flight of messages to the PUT and do not address the PUT statefulness.[1] This excludes vulnerabilities that require at least one round trip where the fuzzer needs to produce testcases made of multiple messages successively fed to the PUT that may depend on outputs the PUT produces as response of previous messages. *(1b) Mutation.* Even for bit-level fuzzers made amenable to protocols, it is overwhelmingly unlikely that bit-level transformations (*e.g.,* random bit flips and basic arithmetic operations) on the network packets express valid and logical transformations over messages such as uses of cryptography to perform *e.g.,* impersonation, downgrade attacks, relay attacks, *etc.* Indeed, the latter often require numerous, structured, and synchronized modifications in the packets. *E.g.,* almost all TLS 1.3 messages are encrypted and integrity-protected[2], which drastically reduce the probability of finding meaningful bit-level mutations (upper-bounded by the negligible probability to break the cryptographic suites). On the contrary, the message as formal term paradigm allows DY fuzzers to produce meaningful message mutations. This *Mutation* limitation affects all fuzzers based on bit-level mutations such as [Fio+20; Zal16; LLV; PBR20; Vra20; boof; Goo; fuzzw; Lee+14; Hoq+16; Mae+22; Som16; Kar+23; PBR20; Nat22; SCP14] or/and based on mutations that can only modify a hard-coded selection of message fields [Lee+14; Mae+22; Som16; Kar+23; SCP14]. We stress that this *Reachability Problem* also concerns memory-related bugs that are solely reachable from deep states, that we can often reach with a DY attacker.

1: See for example the TLS client harness by OpenSSL: https://github.com/openssl/openssl/blob/master/fuzz/client.c

2: Disabling cryptography is not satisfactory as it would only impact the record layer and would hide the attacks related to the use of cryptography.

(2) Detection Problem. Protocol vulnerabilities are **not detected**: the classical spatial and temporal memory errors-related security policies and objective oracle are unable to detect when a protocol vulnerability such as an authentication bypass occurs. That is, even if classical fuzzers reached such an attack state, they would just drop the corresponding test case and deem it uninteresting on the basis that no memory-related error occurred. For the first time, DY fuzzers can detect logical attacks triggering protocol vulnerabilities.

9.1.3.2. Model-based Protocols Fuzzing

Some prior work extended the *stateful fuzzing approach* [Ban+06] and use input-output protocol Finite State Machines (FSM) as a behavioral abstraction of the PUT and use *differential fuzzing* to detect potential bugs [Kar+23; Hus+21] or manually inspect the inferred FSM [RLD22; RP15; Beu+15; Fit+20; PRS15]. They are tailored to capture bypass authentication attacks or other violations of the intended state machine flows. However they are unable to capture the entire class of DY attacks since they cannot tamper with the message contents (except for potentially a finite number of selected/hard-coded values [SCP14; Kar+23]) and thus suffer from the limitation (1b); the same applies to many other testing methodologies and fuzzers [Hoq+17; Sch+22; DPR18; Hus+21; ME04].

Moreover, since the FSM is not specifically designed for security, the security policy violations detected by FSM-based techniques are not necessarily security attacks and require manual inspection, those techniques thus also inherit the *Detection Problem (2)*.

9.1.3.3. Program Verification and Secure Compilation

We argue in [*AHK24*, §B.1] that verification methods targeting implementations of cryptographic protocols (*e.g.,* [Swa+16; Del+21; Del+17; Bha+13; Alm+20; Zin+17; Bha+21; Bau+21; Bar+21a; Ler09; Arq+23]), such as F*, DY*, Jasmin, *etc.* have currently two drawbacks that make them unsuitable to preclude implementation-level logical attacks in *existing* real-world protocol implementations: *scalability* to whole large protocols[3] and, more importantly, ability to *operate on existing, deployed* implementations.

3: See side note 25 in the Introduction of this Part.

9.2. Dolev-Yao Model Abstraction Amenable to Fuzzing

Our fuzzing framework builds on the so-called *DY (DY) model*, going back to the seminal work of Dolev and Yao [DY83] that is today the basis for numerous verification techniques [Bar+21a] we have extensively discussed in the previous Chapters. For the sake of simplicity, we shall introduce a minimal and generic version of the DY model sufficient for our purposes, which is more abstract than the more detailed versions found in previous chapters. Crucially, this abstraction is designed to be amenable to fuzzing through (i) an interpretation of formal terms as bitstrings, which are the actual packets sent and received on the network, and (ii) an instantiation of the protocol agents as implementations thereof we aim to fuzz.

9.2.1. Term Algebra

As already explained in Section 2.3.1, messages are described using a term algebra in the DY model. For example, the term $\mathsf{senc}(m, k)$ represents the message m encrypted using the key k. The algebraic properties of cryptographic functions are specified by equations over terms. For example, $\mathsf{sdec}(\mathsf{senc}(m, k), k) = m$ specifies the expected semantics for symmetric encryption: decryption using the encryption key yields the plaintext. As is common in the DY model, cryptographic messages only satisfy those properties explicitly specified algebraically. This yields the now standard *black-box cryptography assumption*: attackers do not exploit potential

weaknesses in cryptographic primitives beyond those explicitly specified. However, as we shall see, attackers will have complete control over the network. As already mentioned, this allows us to model protocol messages as formal terms from $T_\Sigma(\mathscr{V})$, as defined in Section 2.3.1[4]; see also Examples 2.3.1, 2.3.2 and 2.3.4.

4: There are two main differences with the more detailed definitions found in previous chapters. First, we do not use names and instead work with a finite number of constants, which is sufficient for our purposes. Second, we omit sorts, which boils down to labelling all terms with the same *msg* sort.

Example 9.2.1 (Client Hello) As a TLS related example, consider the ClientHello message. Slightly simplifying (omitting legacy fields and compression methods), we model this message using an operator $\mathsf{CHello}(\cdot,\cdot,\cdot) \in \Sigma$ of arity 3: the term $ch = \mathsf{CHello}(sid, cs, ext)$ represents a ClientHello message where *sid* is a session identifier, *cs* and *ext* are terms that encode the list of proposed cipher suites, respectively extensions. Note that $\mathsf{CHello}(\cdot,\cdot,\cdot)$ models the formatting but does not provide any cryptographic protections. We therefore suppose that we also have three operators π_1, π_2, π_3 that project each of the arguments to allow the DY attacker to extract them. ◊

Example 9.2.2 (Server Hello) A basic model of a MAC, cryptographic hash function as well as a TLS KeyShareEntry (a structure containing the value of a key share and the group it belongs to) can be specified by extending Σ with the operators $\mathsf{MAC}(\cdot,\cdot)$, $\mathsf{checkMAC}(\cdot,\cdot)$, $\mathsf{H}(\cdot)$, and $\mathsf{KS}(\cdot,\cdot)$, and a constant $\mathsf{true} \in \Sigma$ (for valid MAC verification with checkMAC). Slightly simplifying, one could model the ServerHello message with an additional operator $\mathsf{SH}(\cdot,\cdot)$ and a term $sh_0 = \mathsf{SH}(\mathsf{KS}(k, g), p)$ where k models a key share in a group g and p is a PSK identifier, which require other operators not described here. ◊

Remark 9.2.1 Algebraic properties over operators, such as *"verifying a signature with a matching key returns* true*"*, are usually expressed through an equational theory (see Section 2.3.1). Continuing Example 2.3.4, one would specify the equation $\mathsf{checksign}(\mathsf{sign}_t(x), y), \mathsf{pk}(y)) = \mathsf{true}$ with $x, y \in \mathscr{V}$. This does indeed model signature verification as the public key used for verification $\mathsf{pk}(y)$ must match the signature key y, which may be instantiated with any term. To illustrate the *black-box cryptography assumption*, we emphasize that this would be the *only* algebraic property satisfied by checksign, and hence the *only way* to correctly verify a signature in this model is to use a matching key. This models a strong form of the unforgeability cryptographic assumption.
In DY protocol verification, an equational theory is required to *reason about* protocols [CK14; Bla16]. This is not the case in DY fuzzing as operators and terms will be given a bitstring semantics defined by the concrete implementation of the operators in a cryptographic library (see Section 9.3.1). ▷

9.2.2. DY Traces

Despite the *black-box cryptography assumption*, attackers have complete control over the network and the exchanged messages: they can eavesdrop on, inject, and tamper with messages. In particular, such attackers can perform MIM, replay, relay, downgrade attacks, *etc.* Those are examples of *DY attacks*, which are formally defined as the class of attacks that can be triggered by executing a *DY trace* (defined below). The *DY attacker* is an attacker who can perform DY attacks: its behavior is defined as the set of all DY traces. In this Part, we are interested in connecting such an attacker with protocols that are not instantiated as abstract models —*e.g.,* in applied π-calculus— but rather as real-world implementations we aim to fuzz. This requires revisiting some definitions. In the following, we provide a generic definition of DY traces, which allows being instantiated this way.

Syntax Intuitively, *DY traces* are series of networking actions (input and output) a DY attacker can perform. We use the standard notion of *channels* to specify whom the attacker is communicating with. Each channel uniquely identifies an agent. For example, whenever an honest TLS 1.3 client starts a new session it will use a specific channel that the attacker can use to communicate.

Definition 9.2.1 (DY trace [*AHK24*]) *Let $\mathscr{C}$ be a countable set of* channels. *A* DY trace *is a sequence of actions $a_1 \cdots a_n$ such that each action a_i is*

- *either an output* $\mathsf{out}(c, x)$*, where $c \in \mathscr{C}$ and $x \in \mathscr{V}$,*
- *or an input* $\mathsf{in}(c, t)$*, where $c \in \mathscr{C}$ and $t \in \mathscr{T}$.*

The set of traces is denoted by $\mathscr{A}$. ◁

Intuitively, the *output action* $\mathsf{out}(c, x)$ indicates that the DY attacker is waiting to receive a message from the agent on channel c, which will be associated to x. We emphasize that x is not the expected output but a variable that points to the actual message that has been output. The *input action* $\mathsf{in}(c, t)$ indicates that the attacker computes a message and sends it to c, who *inputs* it. This message is obtained by replacing the variables in t by the messages received in the corresponding output actions. Terms in inputs, such as t, are called *attacker terms*[5].

5: Attacker terms are often called "recipes" in the DY model literature.

Example 9.2.3 Continuing Example 9.2.1, we assume two channels c and s of respectively a TLS 1.3 client and server and an adversary acting as a MIM between them. Consider the trace $T := \mathsf{out}(c, x).\mathsf{in}(s, t) \in \mathscr{A}$ which corresponds to: *(i)* the client sends a first message (*e.g.,* a ClientHello) we refer to by x, *(ii)* the term t is then sent by the adversary to the server.

If $t = x$, the attacker simply forwards the server's response to the client. But the attacker can adopt many attack strategies. We give a few examples. *E.g.,* if $t =$ someError, the attacker pretends that the client sent some error message. The attacker could also modify the message referred to by x. Suppose that x points to the ClientHello *ch* defined in Example 9.2.1. Then $t = \mathsf{CHello}(0, \pi_2(x), \pi_3(x))$ would correspond to the same message but replacing the session identifier *sid* by the constant 0. ◊

Example 9.2.4 Continuing Example 9.2.2, consider the trace $T = t_0.\mathsf{out}(c, y).\mathsf{in}(c, sh_0)$. After some prefix trace t_0, the DY attacker, here acting as an attacking server, waits for a message from client c, then sends a valid ServerHello message using term sh_0. One could imagine trying to test a TLS client implementation with a slightly different trace (obtained through *term mutations*) $T^m := t_0.\mathsf{out}(c, y).\mathsf{in}(c, sh^m)$ where $sh^m = \mathsf{SH}(\mathsf{KS}(k, g'), p)$ where we instead use the wrong group in the key share entry to see how the client reacts. A similar trace triggers an RFC violation on wolfSSL we automatically discovered using puffin in DDYF mode [*GHK27*, §5.3.2]. ◊

Semantics We now present generic, formal semantics of DY traces that define how they can be executed. These generic semantics can be instantiated into *DY semantics* (as informally discussed in Remark 9.2.2 and formally defined in Chapter 2 and *e.g.,* in [CK14; Bla16]) or into *concrete semantics*, as done in our *DY fuzzing* approach and detailed in Section 9.3.

Recall that each channel $c \in \mathscr{C}$ corresponds to an honest agent with whom the attacker can communicate. We associate to each channel c the corresponding agent's local *state* s_c and denote by $\mathscr{S}$ the set of all local states. The *global state* is defined by a partial function:

$$\mathsf{s} : \mathscr{C} \rightharpoonup \mathscr{S}$$

which returns this association. We also distinguish the set $\mathscr{S}_0 \subseteq \mathscr{S}$ of *initial* states: when a new agent is created we suppose that it starts in an initial state. We say that a global state s is initial when $\mathsf{s}(c) \in \mathscr{S}_0$ for all $c \in dom(\mathsf{s})$. The attacker's state is a partial function ϕ that associates the variables x of previous output actions $\mathsf{out}(c, x)$ to an abstract notion of messages M, *i.e.,*

$$\phi : \mathscr{V} \rightharpoonup \mathsf{M}.$$

The domain of ϕ contains variables referring to previous outputs from this attacker-state. The set of such partial functions ϕ is denoted by Φ.

We suppose a generic specification of honest agents[6] by the means of

6: Honest agents are the agents we specifically model in DY verification and that we fuzz in DY fuzzing. They are honest in the sense that they follow the protocol specification in the former case or the PUT implementation in the latter case, but they can still be vulnerable to DY attacks.

two abstract partial functions:

$$\begin{aligned} \mathsf{output} &: \quad \mathcal{S} \rightharpoonup \mathcal{S} \times \mathsf{M} \\ \mathsf{input} &: \quad \mathcal{S} \times \mathsf{M} \rightharpoonup \mathcal{S}. \end{aligned}$$

Intuitively, when an agent c is in state s_c, $\mathsf{output}(s_c)$ returns an updated local state s_c' and a message m. Similarly, when a message m is provided to an agent whose local state is s_c, the state is updated to $s_c' := \mathsf{input}(s_c, m)$. Note that those functions are partial as honest agents might block.

In order to transform terms in $\mathcal{T}$ (and notably attacker terms) into messages in M we associate to each operator $\mathsf{f} \in \Sigma$ an *interpretation* $[\![\mathsf{f}]\!] : \mathsf{M}^i \rightharpoonup \mathsf{M}$ when f has arity i. Given $\phi \in \Phi$, we inductively lift $[\![\cdot]\!]$ to terms as follows:

$$\begin{aligned} [\![\mathsf{f}(t_1, \ldots, t_i)]\!]_\phi &= [\![\mathsf{f}]\!]([\![t_1]\!]_\phi, \ldots, [\![t_i]\!]_\phi) \\ [\![x]\!]_\phi &= \phi(x) \qquad \text{if } x \in \mathcal{V} \cap \mathit{dom}(\phi) \end{aligned}$$

We are now ready to formally define how a DY trace can be executed, by the means of a transition system for actions, between pairs of a global state s and an attacker-state ϕ:

$$(\mathsf{s}, \phi) \xrightarrow{\mathsf{out}(c,x)} (\mathsf{s}[c \mapsto s'], \phi \cup \{x \mapsto m\})$$

when $\mathsf{output}(\mathsf{s}(c)) = (s', m)$ and

$$(\mathsf{s}, \phi) \xrightarrow{\mathsf{in}(c,t)} (\mathsf{s}[c \mapsto s_c'], \phi)$$

when $\mathsf{input}(\mathsf{s}(c), [\![t]\!]_\phi) = s_c'$. Intuitively, an action $\mathsf{out}(c, x)$ updates the state of the agent c and records the message m output by c in the attacker's state ϕ. An input action on the other hand provides a message $m := [\![t]\!]_\phi$ as input to agent c and updates the local state of c. The *attacker's computation* of message m is specified by the attacker term t. In particular, the attacker term t may refer to previously output messages using variables in the attacker's state ϕ.

A DY trace $T = a_1 \cdots a_n$ and an initial global state s_0 define an *execution* (starting with an empty initial attacker's state):

$$(\mathsf{s}_0, \varnothing) \xrightarrow{a_1} (\mathsf{s}_1, \phi_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (\mathsf{s}_n, \phi_n).$$

Remark 9.2.2 As is standard in DY protocol verification and explained in Chapter 2, honest agents are usually specified in a formal language such as the applied π-calculus [Bla16] or multiset rewriting rules [Mei+13]. In that case, states in $\mathcal{S}$ correspond to π-calculus processes. More importantly, the messages M are the ground (*i.e.,* without variables) terms —up to the equational theory— with additional private atoms or *names* (that model *e.g.,* secret keys of honest participants) and $[\![\cdot]\!]$ is simply the identity function, *i.e.,* the operators are **uninterpreted**. The formal model yields output that defines which (ground) term can be output by a given process and input that defines the continuation of the process after inputting a (ground) term.

In contrast, as we shall see in Section 9.3.1, for fuzzing we will define a *concrete* semantics. In particular M will be the actual packets (*i.e.,* bit-strings) sent over the network and $[\![\cdot]\!]$ will interpret operators by their actual implementations *e.g.,* in the PUT. We shall also instantiate agents' local states into PUT-specific session handlers (*e.g.,* SSL* pointers in OpenSSL). ▷

Remark 9.2.3 We note that while the interpretation $[\![\cdot]\!]$ transforms terms ($\mathscr{T}$) into messages (M), we do not need to assume that the converse is possible in general, *i.e.,* fully parsing messages as terms. When messages are bitstrings, it may not be possible to parse messages protected by cryptographic primitives. *E.g.,* when h is a hash function, one cannot parse $[\![h(t)]\!]_\phi$ if $[\![t]\!]_\phi$ is not already known. This is why we use a variable x to refer to an output message, to which function symbols can nonetheless, *w.l.o.g.*, be applied by the attacker (through an attacker term t). As we shall see in Section 9.3.1, some partial parsing will nonetheless be possible and applied on output messages in order to improve the way attacker terms refer to some part of messages output by the PUTs. ▷

9.2.3. DY Security Properties

We use the notion of *claims*, which are reminiscent of the *events* in ProVerif or *action facts* in Tamarin, to express security properties. Throughout a trace execution, one can record, in local states, claims that log in which states the honest agents are and what they believe to be their environment (*e.g.,* their peer's identity). *Claims* are expressions $\mathsf{c}(m_1, \dots m_i)$ where c is a symbol with an arity i and $m_1, \dots m_i \in \mathsf{M}$ are messages. We illustrate this notion on two particular kinds of claims that we will use throughout the Chapter. Let $pk, pk_{peer}, m \in \mathsf{M}$.

- *Agreement claims* $\mathsf{Agr}(pk, pk_{peer}, m)$ express that an agent has public key pk and believes to have agreed with a partner having public key pk_{peer} on data m.
- *Running claims* $\mathsf{Run}(pk, pk_{peer}, m)$ express that an agent has a public key pk and believes to be running a session with a partner having public key pk_{peer} and data m.

For example for TLS 1.3, agreement claims will typically be created for clients and servers finishing handshake sessions while running claims are created as soon as m (*e.g.,* session identifier) is available to them. The set of claims is denoted by C. We assume a function $\mathsf{claims} : \mathscr{S} \to \mathscr{P}(\mathsf{C})$ that extracts the set of claims created so far for an agent in a given local state.

Applying the function claims on all local states throughout an execution, we obtain a sequence of sets of claims, called a *trace of claims*. Formally, given an execution

$$(\mathsf{s}_0, \emptyset) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (\mathsf{s}_n, \phi_n)$$

we define the corresponding trace of claims C as $C := C_0, \dots, C_n$ where $C_i = \bigcup_{c \in dom(\mathsf{s}_i)} \mathsf{claims}(\mathsf{s}_i(c))$ corresponds to all claims extracted from the i^{th} state. Claims are therefore *positioned*, and we say $\mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) @ j$ is true in C if $\mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) \in C_j$. The logic for expressing security properties is the same as the one used in DY verification, which was introduced in Section 2.3.3.

Example 9.2.5 *Non-injective agreement* on some data m is the property:

$$\forall \mathsf{pk}, \mathsf{pk}', m, i.\ \mathsf{Agr}(\mathsf{pk}, \mathsf{pk}', m) @ i \Rightarrow \exists j.\ \mathsf{Run}(\mathsf{pk}', \mathsf{pk}, m) @ j$$

Intuitively, whenever an agent identified by pk believes they successfully agreed with agent pk′ on m then agent pk′ indeed started a session with pk on data m. ◊

9.3. DY Fuzzing Design

At a high level, we propose with *DY fuzzing* to use the DY attacker and DY traces as domain-specific knowledge to produce test cases and detect DY attacks.

The **search space**, *i.e.,* the set of all *test cases*, is the set of all DY traces $\mathscr{A}$. Starting from a *Seed Corpus*, DY traces are **mutated** and then executed

on a PUT. To execute a DY trace, we rely on two components.

- The *Mapper concretizes* terms in $\mathcal{T}$, and notably attacker terms, into bitstrings, *i.e.,* it computes $[\![\cdot]\!]$. It also implements a partial parsing function $\lfloor\cdot\rfloor$ to extract some structure from bitstrings output by the tested PUT and produce sub-messages (*e.g.,* fields of some structure) that are included in the attacker's state ϕ and that can then be used in attacker terms.
- The *Harness* sends those adversarial bitstrings $[\![t]\!]$ to the PUT's agent sessions associated to channels in $\mathscr{C}$. The PUT sends back bitstrings that get parsed through $\lfloor\cdot\rfloor$ and added to ϕ. Therefore, the set of messages M is the set of bitstrings, and the output and input functions are implemented through the interaction of the *Harness* with the PUT. The *Harness* also observes the execution and extracts claims (*i.e.,* function claims(·)) from the PUT's internal states.

These claims and other execution metadata are then analyzed by the *Objective Oracle* that detects security policy violations, in particular DY property violations. As is standard, the *Harness* is PUT-dependent. However, we emphasize that the *Mapper* and *Objective Oracle* are only *protocol*-dependent and PUT-independent.

Benefiting from our generic presentation of the DY model in Section 9.2, we specify those different components in the next subsections. We remain as generic as possible: the idea of DY fuzzing is applicable independently of the protocol and the PUT, provided that it implements a cryptographic protocol. We implemented this generic design in PUFFIN. We sometimes exemplify some aspects with our main instantiation, TLSPUFFIN, that implements a DY fuzzer for TLS against various TLS libraries (see Section 10.1). The overall fuzzing loop is summarized in Section 9.3.6 and depicted in Figure 9.2.

9.3.1. Mapper

The *Mapper* is responsible for connecting the formal term abstraction used in DY traces with the concrete semantics of the PUTs, which receive and send bitstrings. When multiple PUTs implementing the same protocol are tested, the same *Mapper* can be reused. Hence, the *Mapper* is PUT-independent and can be written once per protocol, as we have done for TLSPUFFIN.

Concretization In particular, the *Mapper* is primarily used to *concretize* terms into protocol messages as bitstrings that can be sent to the PUT. Formally, it implements $[\![\cdot]\!]$: for any $f \in \Sigma$, it computes $[\![\mathsf{f}]\!] : \mathsf{M}^i \rightharpoonup \mathsf{M}$ where M is the set of bitstrings ($[\mathrm{u8}]$). One way to implement the mapper is to use the PUT's implementation of a primitive f to compute the concretization $[\![\mathsf{f}]\!]$. However, a different implementation (*e.g.,* a reference or a custom implementation) could also be used. For instance, in TLSPUFFIN, we reused part of the Rust library RUSTLS that implements TLS to compute $[\![\cdot]\!]$ for the 240 operators of the signature we used.[7] This signature is not exhaustive but contains the symbols necessary to produce terms corresponding to all types of messages of TLS 1.2 and 1.3, making it possible to run full TLS sessions in various modes (see Section 9.3.3).

Parsing and Knowledge Extraction While "re-interpreting" bitstrings to terms is not always possible (see Remark 9.2.3), one can *partially* parse protocol messages until reaching some opaque data such as cipher

7: Note that the more symbols there are in the term algebra's signature, the more capabilities the adversary has to construct attacker terms.

Remark 9.3.1 We make sure to use an unequivocal representation of data as bitstrings; *e.g.,* we use the DER format for certificates. Note that $[\![\mathsf{f}]\!]$ is a partial function since its computation might fail or return an error. For an constant f_0, we define $[\![\mathsf{f}_0]\!]$ as the corresponding, statically generated, data item. For instance in TLSPUFFIN, we statically generate the RSA public keys for a bounded number of agents and then bind each to a constant in the term algebra. Some of those agents are assumed to be compromised (in control of the adversary). For those, we also include the corresponding RSA private key in the term algebra so that the attacker can use them in attacker terms (in inputs). ▷

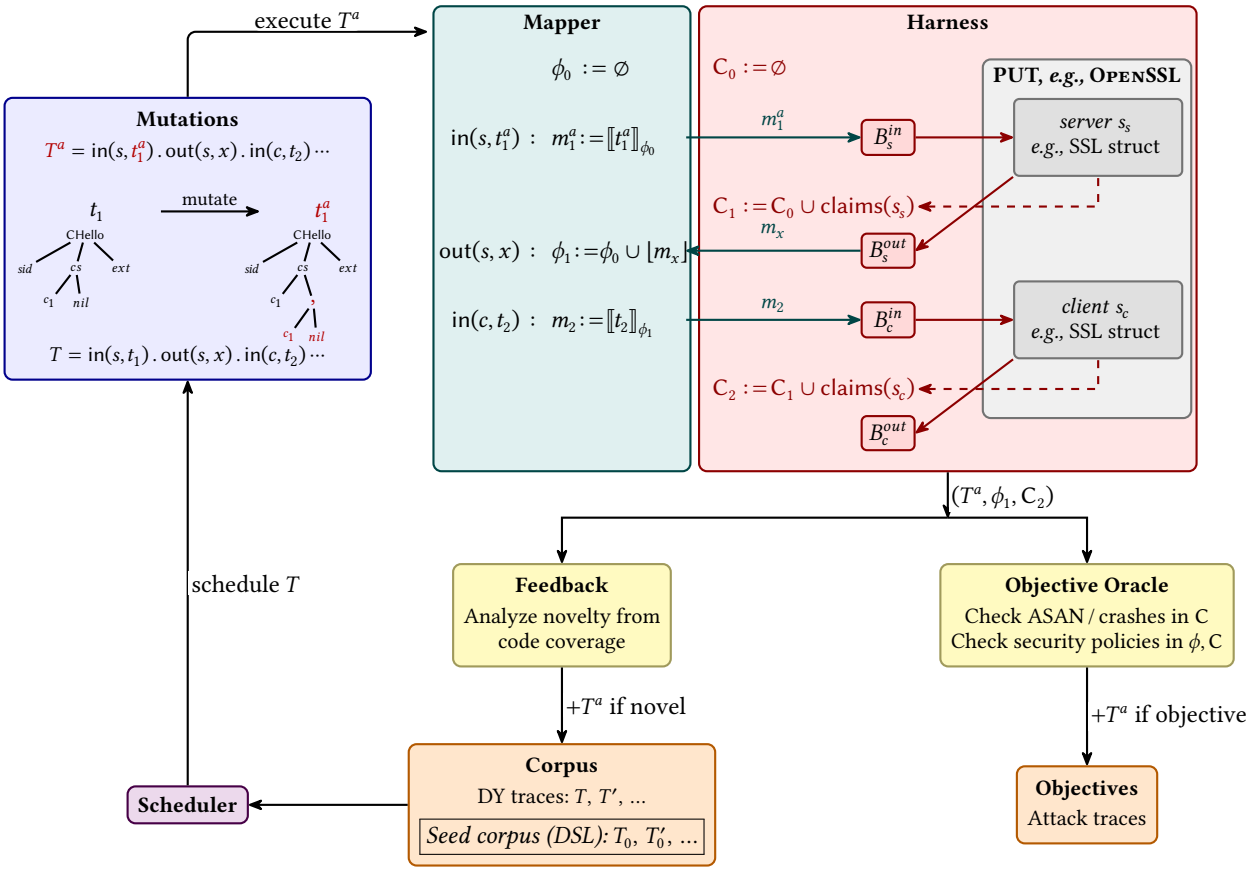


Figure 9.2.: Overview of the DY fuzzing loop (DY-specific instantiation of Figure 9.1). Starting from a *Seed Corpus* of symbolic DY traces, the *Mutation* stage applies term-level and action-level mutations to produce a new trace T^a. The *Mapper* concretizes each attacker term into a bitstring under the current attacker's state ϕ (*e.g.*, $m_1^a = [\![t_1^a]\!]_{\phi_0}$) and, after the PUT responds, parses the output message to extend ϕ. The *Harness* manages the PUT instances (here a server s_s and a client s_c, each with its own SSL struct) and uses input/output buffers B^{in}/B^{out} to exchange messages with them. Note that the same agents and their buffer pairs is reused across all protocol steps of a given trace (*e.g.*, the same B_s^{in} is used for all messages inputs to the server s_s), but are re-initialized before executing a new trace. After each action execution, claims C are extracted from the PUT internal state to capture security-relevant information (*e.g.*, negotiated parameters). The *Feedback* component evaluates the novelty of the execution based on code coverage, while the *Objective Oracle* checks for memory errors (via ASAN) and for violations of DY security policies over ϕ and C.

texts, hashes, or signatures. We thus assume a *partial parsing function* $\lfloor \cdot \rfloor : \mathsf{M} \rightharpoonup (\mathscr{T}, \sigma)$ that extracts as much structure as possible from a bitstring and returns a term t along with a substitution σ mapping the variables of t to bitstrings, which cannot be parsed further.[8]

Each time an output is executed and a new message $m \in \mathsf{M}$ is received from the PUT, it is automatically parsed using $\lfloor \cdot \rfloor$ yielding some (t, σ). All sub-terms of t, including t, are then interpreted as bitstrings $([\![t']\!]_\sigma$ for t' a sub-term of t) and stored in the attacker-state ϕ, associated with fresh variables. We call these "extracted" messages *Knowledge*, which are elements of M (bitstrings). As we shall see in Section 9.3.3, these *Knowledges* can then be queried and used in attacker terms when writing DY traces.

8: Note that for a ground and "plain text" term t, one has $\lfloor [\![t]\!]_\emptyset \rfloor = (t, \emptyset)$.

Example 9.3.1 A TLS ServerHello bit-string $m \in \mathsf{M}$ could be parsed as $\lfloor m \rfloor = (sh, \sigma)$ with $sh = \mathsf{SH}(\mathsf{KS}(x, y), z)$, $\sigma = \{x \mapsto m_k, y \mapsto m_g, z \mapsto m_p\}$, where m_k is a key share, m_g a group, and m_p a list of PSK ids (this is simplified as these fields are structured themselves). The produced knowledges are m, m_k, m_g, m_p as well as $[\![\mathsf{KS}]\!](m_k, m_g)$, *i.e.*, the bit-string representing the key share term $\mathsf{KS}(x, y)$. ◊

9.3.2. Harness

As is standard, a PUT-specific *Harness* is required. While it could be possible to create a *Harness* that just passes a single protocol message to an agent, it would fail to reach parts of the code only accessible after several flights of messages and potential local state updates. DY fuzzing neatly addresses this with a *Harness* that executes DY traces on the PUT as explained next. Given a test case, that is a DY trace $T = a_1 \cdots a_n \in \mathscr{A}$, it will do the following.

Agents creation The *Harness* creates a new PUT session s_c for each channel $c \in \mathscr{C}$ that appears in T. The *session handler* of each of these sessions then corresponds to the *local state* of the agent identified by c in our generic DY model. Those sessions have a notion of *input buffer* and *output buffer*, where bitstrings can be read, respectively written, as well as a notion of progress: we assume a function progress can be called on a session to instruct the corresponding agent to read and process a message on the input buffer and possibly write a message on the output buffer. Configurations of those sessions, and thus their initial states $s_c \in \mathcal{S}_0$, are those of the channels (*e.g.,* client vs. server). The initial global state s_0 is composed of all those s_c.

For example for the OpenSSL implementation of TLS 1.3, the *Harness* creates new SSL objects (pointers of type SSL* storing a server or client session state) for each channel in T. The progress function is then `int SSL_do_handshake(SSL *ssl)`.

Communication The actions of the trace T are executed according to the transition relation $\xrightarrow{a}$ (see Section 9.2.2) instantiated as follows. First, $[\![\cdot]\!]$ is computed by the *Mapper* (see Section 9.3.1) on attacker terms. Second, we shall instantiate the input and output functions as follows. The state $s_c' := \mathsf{input}(s_c, m)$ is obtained by first writing m into the input buffer of the current state s_c and then calling the progress function that modifies the state to s_c'. Next, $(s_c', m) := \mathsf{output}(s_c)$ is computed by reading m from the output buffer of the current state s_c; the resulting state s_c' is identical to s_c up to the output buffer.

Claim extraction The *Harness* is also responsible for extracting claims from agents, that is it implements a function $\mathsf{claims} : \mathcal{S} \to \mathscr{P}(\mathsf{C})$ (see Section 9.2.3). In general, doing so may depend on the PUT and the possibility to introspect the agents' internals but is often straightforward to do because required data are usually exposed. We detail two different approaches to do so in Section 10.1.2.

If the PUT offers no or no easy introspection (*e.g.,* the PUT is closed-source), it is still possible to extract some claims solely based on the DY trace being executed and the attacker's state. For instance, when a PUT server returns a Server Finished, it is possible to infer that the server believes it has finished the handshake.[9]

9: For instance, a TLS 1.3 client makes a Agr(·) claim upon reception and acceptance of a server Finished message; to set the claim's arguments the client's public key is the one used during client session initialization, and the peer's public key could be extracted from the previously received server CertificateVerify message. TLSPUFFIN did not use this since introspection was possible and relatively easy to implement.

9.3.3. DY Traces and Seed Corpus

As mentioned, test cases are DY traces, and the search space is $\mathscr{A}$. The seed corpus is a set of DY traces that are used as starting points for mutations.

Mutation	Description
Skip	Removes an action from a trace
Repeat	Repeats an action from the trace to the trace
Swap	Swaps two (sub-)terms in the trace
Generate	Replaces a term by a random one
Replace-Match	Swaps two operators in the trace
Replace-Reuse	Replace a (sub-)term by another (sub-)term in the trace
Remove-and-Lift	Replaces a (sub-)term by one of its sub-terms

Table 9.1.: Summary of action-level and term-level mutations for DY fuzzing.

DY Traces and Queries The user needs to write DY traces to be used as seeds for the fuzzing campaign. We explain in Section 10.1.2 how we designed and implemented a Domain Specific Language (DSL) to ease this task. However, we had quickly noticed that the way the attacker refers to its state ϕ was often not robust enough through successive mutations.[10] We alleviate this issue with *queries*. When adding to ϕ all *Knowledge*s obtained by partially parsing PUTs' outputs with $\lfloor\cdot\rfloor$, we also store from which agent $c \in \mathscr{C}$ it originates, the kind of message it is (*e.g.,*ClientHello for TLS) and other metadata such as the message type. A query is essentially a conjunction of conditions over metadata (*e.g.,* agent c, protocol message kind, message type) to access the first matching message in the attacker's state ϕ. It also eases the writing of the seed traces described next.

10: *E.g.,* consider a variable x and an attacker term $t = \mathrm{sdec}(x, k)$ in a trace $T = T_1.\mathrm{out}(c, x).\mathrm{in}(c', t)$. When executing this trace T, x is assigned a message that can indeed be decrypted with k. Now, after some mutations affecting T_1, the agent c might not send an encrypted message anymore but *e.g.,* an error message. Yet, the attacker term t remains the same and its computation will now fail. Instead of using x to refer to the ciphertext sent by c, it would be more robust to use a query looking for the first AppData message sent by c.

Seed Corpus We consider a bounded number of agents (and thus of channels), enough to be able to express all possible *happy flows*[11] for all possible expected protocol configurations. When evaluating our fuzzer we solely use happy flows and no attack traces as seeds.[12]

For instance for TLS 1.3, we consider two different honest agents, a client Alice and a server Bob. We consider seeds for each of the following happy flows: *(i)* full handshake between Alice and Bob where the DY attacker acts as a passive MIM, *(ii)* full handshake between the DY attacker, acting as an honest server, and Alice acting as client, *(iii)* full handshake between the DY attacker, acting as an honest client, and Bob acting as server, and *(iv)* the happy flow of item *(iii)*, followed by a resumption handshake with the same agents. This yields 15 seeds in total for TLS (1.2 and 1.3).

11: We call *happy flow* an honest and expected message flow. If the adversary is a MIM, then it forwards messages without modifying them. If it acts as a server or client, it behaves as an honest one.

12: Otherwise, attacks would always be found immediately and bias the evaluation. For real fuzzing campaigns, including attack traces on previous versions of the PUT, or stemming from a different implementation, is however desirable as it allows for regression testing and may also ease finding variants of an attack that was not properly fixed.

9.3.4. Mutations

Since the *Seed Corpus* captures all relevant execution scenarios for a given fuzzing campaign, we only consider mutations that do not create new channels. However, mutations can modify the structure of the trace at the *action-level* (*e.g.,* swapping two actions, dropping a message, *etc.*). They can also modify the content of the attacker terms at the *term-level* (*e.g.,* replace a sub-term by another one to express a credential swapping, add a sub-term to add a TLS 1.3 extension that was inexistent, *etc.*). We describe all mutations we consider for TLSPUFFIN in Table 9.1 (split into action- and term-level mutations). We consider these to be a good basis for any DY fuzzer as they fully capture the DY attacker and are completely protocol (and obviously PUT) independent. See an ablation study and why each single mutation is useful in [*AHK23*, §C.1.5].

Action-level mutations The *Skip* mutation removes a random action from a trace. The *Repeat* mutation repeats an action: a random action in the trace is copied and inserted at a random new position (and in case of an output actions, the output variable is renamed). These two action-level mutations are already enough to capture some authentication bypasses such as the ones from [Beu+15] and **SKIP** from Table 10.2.

Term-level Mutations A major advantage of DY fuzzers is their ability to also mutate attacker terms and thus deeply change the structure of exchanged messages, even under cryptographic protections such as encryption (*e.g.,* replace, remove or add TLS extensions), and/or modify very specific fields possibly using cryptographic primitives (*i.e.,* for expressing a certificate swapping). These mutators require a description of test cases that specifies the structure of messages and the available cryptographic primitives, which is one of the main novelty of our DY fuzzing approach. They all start by randomly picking an input action $\mathsf{in}(c, t)$ and then mutate the attacker term t. For example, the *Replace-Match* mutation replaces an operator $f \in \Sigma$ in t with a different one $f' \in \Sigma$ of the same arity as f. This can be seen as changing the implementation of some computations (*e.g.,* replacing SHA2 with SHA3) or changing the values of some atoms (*e.g.,* swapping Alice's public key with Bob's public key). Another example is the *Remove-and-Lift* which chooses at random a subterm t' of t and replaces it with a random sub-term t'' of t'. In particular, this mutation can remove random elements from a list.

Example 9.3.2 Suppose we have an initial trace $T = \mathsf{in}(s, ch).\mathsf{out}(s, x)$ where $ch = \mathsf{CHello}(sid, cs, ext)$ as in Example 9.2.1. This models the case where a server receives a ClientHello from an attacking client before responding with a term t. Suppose that cs and ext are nil terminated lists and contain, respectively a single cipher suite c and a list of n extensions $e_1, \ldots, e_n$. Then, the attacker can

- remove extension e_n by $ch_1 := \textit{Remove-and-Lift}(ch)$:
  $ch_1 = \mathsf{CHello}(sid,\ [\mathrm{c}, \mathrm{nil}],\ [e_1, \cdots, e_{n-1}, \mathrm{nil}])$
- add a cipher suite c by $ch_2 := \textit{Replace-Reuse}(ch_1)$:
  $ch_2 = \mathsf{CHello}(sid,\ [\mathrm{c}, \mathrm{c}, \mathrm{nil}],\ [e_1, \cdots, e_{n-1}, \mathrm{nil}])$
- iterate *Replace-Reuse* k times and obtain ch_k where c is repeated k times: $ch_k = \mathsf{CHello}(sid,\ [\mathrm{c}, \cdots, \mathrm{c}, \mathrm{nil}],\ [e_1, \cdots, e_{n-1}, \mathrm{nil}])$

Sending ch_k to the server triggers a HelloRetryRequest message due to the missing extension e_n, *i.e.,* when removing the supported_groups extension. Then applying the *Repeat* mutation, we obtain the trace $T^a = \mathsf{in}(s, ch_k).\mathsf{in}(s, ch_k).\mathsf{out}(s, x)$. We shall see in Section 10.2.1.4 that additional mutations, notably involving cryptographic operators, lead to a buffer overflow on wolfSSL when $13 \leq k \leq 150$. ◊

Mutation Constraints Mutations can be applied only when specific constraints are fulfilled. In particular, we shall exclude mutations that yield traces that cannot be gracefully executed (through $[\![\cdot]\!]$) for example because of an ill-formed attacker term. We thus restrict to well-typed attacker terms according to the type system of the library we use to build the *Mapper*. We also impose sane limits on the number of actions and attacker term sizes. We provide more details in [*AHK23*, §D.2.1].

9.3.5. Security Policies and Objective Oracle

The *Objective Oracle* observes executions made by the *Harness* and looks for security policy violations. When such a violation is detected, the corresponding test case is flagged as an objective and is stored to disk.

Memory-related *Objective Oracle* Fuzzing is classically used to discover memory related bugs. Those bugs are easy to detect by relying on operating system signals or code instrumentation techniques such as ASAN [Ser+12]. DY fuzzers *also* consider such a security policy (*i.e.,* no memory-related bugs) and *Objective Oracle* (code instrumentation with ASAN) but we strive to go beyond. Indeed, as such, the fuzzer could encounter a protocol vulnerability and would never raise any alarm; the user would simply never know and would miss them.

DY properties as policies To remedy this problem, we consider DY properties (Section 9.2.3) as additional security policies. For example, for TLS 1.3 and TLSPUFFIN, we consider the DY properties corresponding to mutual agreement on handshake data as policies in addition to using ASAN. We next explain how one can translate DY properties into an operational *Objective Oracle* detecting them.

***Objective Oracle* for DY properties** As explained in Section 9.3.2, the *Harness* provides a function $\text{claims} : \mathcal{S} \to \mathcal{P}(\text{C})$. Therefore, a trace of claims can be extracted throughout the execution by the *Harness* and the *Objective Oracle* can evaluate the validity of all the considered DY properties at any execution step, according to the properties semantics defined in Section 9.2.3. As soon as one DY property is falsified, the *Objective Oracle* flags the corresponding test case as an attack candidate.

9.3.6. Big Picture: The DY Fuzzing Loop

Each fuzzing campaign starts with a *Corpus* initialized to the *Seed Corpus*. The main fuzzing loop depicted in Figure 9.2 proceeds as follows: a DY trace $T \in \mathscr{A}$ is picked from the current Corpus, multiple mutations are applied yielding $T^a \in \mathscr{A}$. The *Harness* executes T^a on the PUT and, when necessary, calls the *Mapper* to concretize all attacker terms in input actions. It also collects feedback (*e.g.,* code-coverage in the PUT through code instrumentation) and observations (claims and potential ASAN errors). If the *Objective Oracle* identifies a security policy violation on those observations, notably a DY property violation, it flags T^a as an attack trace and stores it. Otherwise, based on the achieved code coverage, the fuzzer decides whether T^a is worth being stored in the *Corpus*. The fuzzer then proceeds to the next loop iteration.

9.4. Brief Overview of Differential DY Fuzzing

We now briefly present the design of Differential Dolev-Yao Fuzzing (DDYF), introduced in [*GHK27*]. Recall that DYF relies on manually written security policies checked by the *Objective Oracle* to detect attacks. Defining these policies is labor-intensive and inherently non-exhaustive: only a subset of logical attacks is detected in practice. DDYF addresses

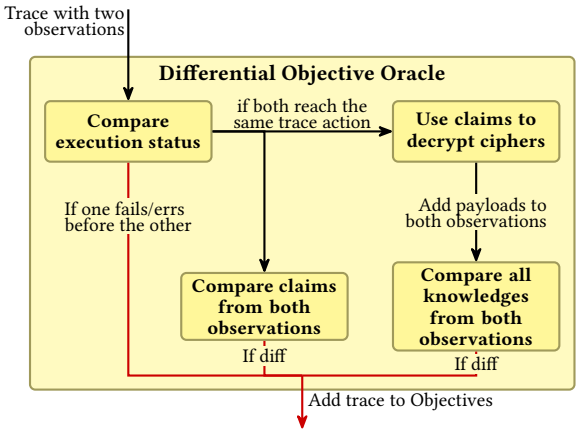


Figure 9.3.: Differential Objective Oracle workflow

this limitation by replacing the DY security oracle with a *Differential Oracle* that compares executions of a same DY trace on two different PUTs implementing the same protocol and flags discrepancies as potential bugs. We refer to [*GHK27*] for full details and focus here on the key ideas, challenges, and solutions.

9.4.1. Architecture and Differential Oracle

DDYF modifies several components of the DY fuzzing architecture (Figure 9.2). First, the *Harness* is extended into a *Differential Harness* that executes and observes two PUT instances in parallel (see Figure 9.4). The two *Observers* collect for each PUT execution: the execution status (*i.e.,* success or error, and the last successfully executed trace action), the *Knowledges* (parsed output messages), and the *Claims* (PUT state information). Second, the *Feedback* uses the code coverage of the union of both PUTs as feedback metric. Third, the *Objective Oracle* is replaced with a *Differential Oracle* (Figure 9.3) that analyzes observation differences and flags objective traces.

The *Differential Oracle* compares three types of observations, each targeting different classes of bugs; see Figure 9.3.

1. **Execution status.** Comparing the execution status and last executed action detects when one PUT rejects a message while the other accepts it and progresses further, as typical with denial of service or authentication bypass bugs. The oracle reduces each execution to *Success* or *Error* (with the action where execution stopped) and flags a difference when one PUT stops before the other or their statuses differ.
2. ***Knowledges* (exchanged messages).** Comparing the *Knowledges* (*i.e.,* parsed messages stored by the fuzzer) detects parameter mismatches between PUTs, typical of downgrade attacks (*e.g.,* different negotiated cipher suites). As we shall see, this comparison requires significant care to avoid false positives due to randomness, encryption, and specification permissiveness.
3. ***Claims* (PUT internal state).** Comparing claims detects state-machine flaws. Since *Claims* are extracted through PUT-specific callbacks whose timing and multiplicity may differ across implementations, claims are deduplicated and compared separately per agent, independently of the action at which they were produced.

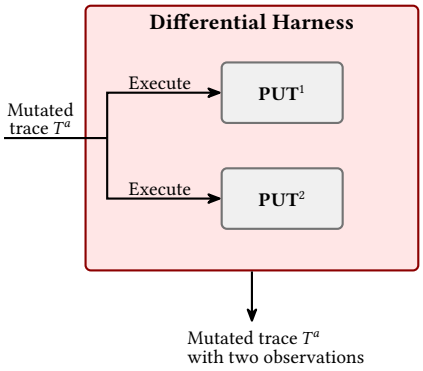


Figure 9.4.: Differential Harness Architecture for two PUTs. The execution of the trace on each PUT follows the same workflow as in Figure 9.2 but with two parallel executions and observations.

9.4.2. Key Challenges and Solutions

Comparing cryptographic protocol executions raises several challenges that we now discuss along with the solutions from [*GHK27*]. The DY model underpinning DDYF is the key enabler for all of them.

(1) Non-determinism Cryptographic protocols inherently produce non-deterministic outputs due to randomness (*e.g.,* nonces, key shares) that propagates through cryptographic operations to whole messages. Comparing exchanged messages as bitstrings would thus lead to systematic false positives; moreover, using a uniform PRNG seed is insufficient since different PUTs derive random values differently. The solution is to re-interpret messages as DY terms and *filter* (blacklist) randomized fields from the comparison while preserving attacker capabilities (*i.e.,* the attacker can still *use* blacklisted knowledges but the oracle does not *compare* them). For TLS, 24 out of 145 structure fields are blacklisted. The blacklist is identified through a simple semi-automated methodology: running a differential campaign of a PUT against itself so that all observed differences stem from randomized values.

(2) Encryption Most cryptographic protocol messages are encrypted; simply ignoring ciphertexts would miss most of the handshake. For instance, in TLS 1.3, all messages following ServerHello are encrypted. The DY fuzzing framework provides the necessary ingredients to overcome this: ciphertexts are stored as *Knowledge*s, decryption keys are accessible through *Claim*s, and encryption/decryption are already operators in the DY term algebra. We write, once per protocol, a few *decryption recipes* (*i.e.,* terms in the DSL specifying how to decrypt given ciphertexts using the appropriate key material). The oracle evaluates these recipes after trace execution and before knowledge comparison; the resulting plaintexts are treated as new output messages from which *Knowledge*s are extracted. For TLS, only two decryption recipes suffice. We emphasize that the decryption keys obtained through *Claim*s are only used by the oracle, not exposed to the attacker.

(3) Specification permissiveness Protocol specifications may allow multiple valid syntactic representations for semantically equivalent actions. For instance, TLS allows server extensions to be sent as four separate encrypted records or coalesced into a single encrypted record. Syntactically comparing these would lead to false positives. Two solutions are combined. First, new operators are added to the DY term algebra to abstract over syntactic variants. For TLS, we introduced MessageFlight and OpaqueMessageFlight types with associated encryption/decryption operators that transparently return the same list of payloads regardless of whether the input is a single coalesced ciphertext or a list of individual ciphertexts. Second, custom comparison routines for some protocol sub-message types absorb benign syntactical differences (*e.g.,* treating extension lists as multisets rather than lists, since their order is unspecified). Both are implemented through lightweight code annotations.

(4) PUT configuration differences Different PUTs may have different default configurations (*e.g.,* supported ciphers, extensions, cipher preference order). We mitigate this by identifying a protocol-specific common set of parameters supported by both PUTs and aligning them, preferring

API-provided runtime configuration to preserve the default (most commonly deployed) build.

(5) Triaging As with any fuzzer, triaging is crucial to prioritize and assess differences. This is exacerbated in differential fuzzing due to a high number of objectives where benign differences are mixed with potentially serious vulnerabilities. We designed a two-step semi-automated triaging methodology [*GHK27*]. A first step groups objective traces into *buckets* with similar differences to deduplicate and prioritize them, using automated scripts and a dedicated `diff-execute` CLI tool (producing human- and machine-readable difference reports). A second step investigates each bucket to identify root causes, iteratively splitting buckets until each corresponds to an isolated behavior.

10. Implementation, Evaluation, and Case Studies

We first present our implementation of DYF and DDYF within the PUFFIN family of fuzzers. We then describe our evaluation methodology and present both qualitative and quantitative results that highlight the characteristics of DY fuzzing —particularly its effectiveness in identifying DY attacks. All of these experiments are reproducible using scripts available at [puffin] and are described in detail in [*AHK24*; *GHK27*]. Next, we discuss our evaluation of Differential Dolev-Yao Fuzzing (DDYF) in Section 10.3. Finally, we conclude with a summary of our findings and a responsible disclosure timeline for the vulnerabilities and bugs we identified in Section 10.4.

10.1 The PUFFIN DYF Library and its Fuzzers 171
10.1.1 Modular Architecture . . 172
10.1.2 Implementation Challenges and Features . . . 173
10.1.3 Implementing Differential DY Fuzzing in PUFFIN 174
10.2 Evaluation of DY Fuzzing 174
10.2.1 Finding DY Attacks . . . 174
10.2.2 Coverage Comparison . 178
10.3 Evaluation of Differential DY Fuzzing 180
10.4 Summary of Findings and Responsible Disclosure 183

10.1. The PUFFIN DYF Library and its Fuzzers

We present PUFFIN and its derivatives, which together form a public FLOSS framework† written in RUST. We notably contribute a generic RUST library for building DY fuzzers (PUFFIN) for arbitrary protocols and a full-fledged DY fuzzer for TLS (TLSPUFFIN) using PUFFIN. We used TLSPUFFIN to fuzz OPENSSL, BORINGSSL, LIBRESSL, and WOLFSSL. To show the modularity and generality of PUFFIN, we also built SSHPUFFIN for SSH and a preliminary PoC for OPC UA (OPCUAPUFFIN).

As of version 2.0 of PUFFIN and TLSPUFFIN (and the dev branch for SSHPUFFIN and OPCUAPUFFIN), we have written ca. 58.8k LoC for this project (excluding dependencies), 48.2k of them in RUST and most of the rest C harness code: 17.2k for PUFFIN (core engine, build infrastructure, and generic mapper interface), 13.4k for TLSPUFFIN (protocol instantiation and TLS mapper) and 6.1k for its 4 harnesses, 9.5k for OPCUAPUFFIN (in total), and 9.7k for SSHPUFFIN (in total, including its LIBSSH and WOLFSSH harnesses), complemented by 2.9k of external verification tests. Of these, 38.9k are implementation code, 9.2k tests, and 10.7k boilerplate, mostly hand-written seed and attack traces. PUFFIN and TLSPUFFIN alone have grown about 2.5-fold since the ca. 16k LoC of the version presented in [*AHK24*], while the whole project has grown about 3.7-fold. Table 10.1 provides the detailed breakdown.

†: The code is accessible on our GitHub repository [puffin] (https://github.com/tlspuffin/tlspuffin) and the user and developer documentation is available on the PUFFIN website https://tlspuffin.github.io/.

Category	Impl.	Test	Boiler.	Total
PUFFIN Core Engine	11,085	1,724	0	12,809
PUFFIN Build Infrastructure	1,729	24	0	1,753
PUFFIN *Mapper* Interface	1,895	788	0	2,683
TLSPUFFIN *Mapper*	3,865	348	243	4,456
TLSPUFFIN Core (Protocol)	2,615	1,400	4,898	8,913
TLS Harnesses (for the 4 PUTs)	6,102	0	0	6,102
OPCUAPUFFIN (all)	5,318	1,072	3,131	9,521
SSHPUFFIN (all)	6,290	941	2,429	9,660
Verification (External Tests)	0	2,874	0	2,874
TOTAL (CONTRIBUTED)	**38,899**	**9,171**	**10,701**	**58,771**

Table 10.1.: LoC analysis of the PUFFIN library and its family of fuzzers (version 2.0; dev branch for SSHPUFFIN and OPCUAPUFFIN). All counts are as reported by cloc, excluding comments, blank lines, configuration files, and external RUST dependencies. For code we copied from upstream libraries and then modified (mainly for writing *Mappers*), only the lines we added or changed are counted. The *Test* column covers unit tests and test utilities. The *Boiler.* column isolates repetitive code: seed and attack traces and the declarative function signatures of the *Mappers*. *Build Infrastructure* fetches, patches, instruments (coverage, sanitizers), and builds the PUTs, and links their harnesses into the fuzzers.

10.1.1. Modular Architecture

Our project features a modular design that facilitates reuses. We present its main modules (aka Rust *crates*) and their interfaces.

puffin: our DY fuzzing library puffin is a generic Rust library to build DY fuzzers. It is protocol- and PUT-agnostic: it defines a minimal interface (aka *traits*) for a protocol and its security properties as well as for the PUT-harness; see below. Given a crate implementing this interface (*e.g.,* tlspuffin for TLS and the 4 aforementioned PUTs), puffin implements a DY fuzzer for them.

puffin builds on the state-of-the-art LibAFL [Fio+22] fuzzer library in order to implement an *evolutionary* fuzzing loop. We implement custom *Harness* interface, *Mutator*, and *Objective Oracle*, as well as an additional *Mapper* interface component. For this, we implement a generic term algebra amenable to the fuzzing setting: terms must be serializable, executable (*i.e., Mapper*), introspectable for the mutations (see Section 10.1.2), *etc.* Similarly, we implement generic DY traces that can be executed on any given PUT (*Harness*). Since the traces of the *Seed Corpus* must be written by hand, we made puffin offer a DSL for declaratively defining DY traces (see [*AHK23*, §D.1]). We use the standard AFL-like code edge coverage map [Fio+22] (*i.e.,* hit counts) as feedback metric. We implemented all mutations from Table 9.1 resulting in a generic *Mutator*. Finally, all given DY properties are checked throughout the execution by our custom *Objective Oracle*.

Using puffin, one can launch a fuzzing campaign on a given PUT, which is the main use case, but also execute a given trace (*e.g.,* an objective, or a custom trace written with our DSL) on a given PUT, produce a graphical representation of a trace, or compute the packets agents send in a trace (using $[\![\cdot]\!]$ and the *Mapper*). We implemented all those features, which ease bug triaging (see Section 10.1.2).

puffin interfaces Integrating a new protocol into puffin reduces to implementing a single Rust trait (ProtocolBehavior, see Figure 10.1) that notably fixes:

- the protocol's message ontology (*structured* parsed messages, *opaque* bitstring messages, and their *flights*),
- the *Mapper*: available operators and how to encode and partially parse terms to and from bitstrings ($[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$),
- the protocol-level *claims* emitted by the PUT,
- the *security violation policy* defined over claim sequences,
- the types of protocol messages used in knowledge queries inside traces, and
- a set of DY traces for seeding the corpus.

The rest of puffin (fuzzing loop, term algebra, mutators, *Objective Oracle*, DSL, coverage feedback, *etc.*) is generic over this trait.

A new PUT is in turn plugged through a minimal C ABI interface (AGENT_INTERFACE, see Figure 10.2) that puffin drives as a state machine, with streaming I/O, state-progress and reset operations, and a claim callback by which the PUT surfaces security-relevant internal events (*Claims*) to our *Objective Oracle*. A new PUT harness is thus a thin wrapper around the underlying PUT library's API, that the Rust Cargo build system compiles and links with puffin through this C interface. Note that we have explored the use of Large Language Models (LLMs) for

```rust
pub trait ProtocolBehavior {
  type ProtocolTypes;
  type Claim;
  type SecurityViolationPolicy;
  type ProtocolMessage;
  type OpaqueProtocolMessage;
  type ProtocolMessageFlight;
  type OpaqueProtocolMessageFlight;

  // Initial seed corpus of DY traces.
  fn seed_corpus(put: PutDescriptor)
    -> Vec<(Trace, &'static str)>;

  // Encode term as bitstring.
  fn concretize(m: &dyn EvaluatedTerm)
    -> Vec<u8>;

  // Parse bitstring to typed message.
  fn partial_parse(bs: &[u8], ty: TypeId)
    -> Result<Box<dyn EvaluatedTerm>,
        Error>;
}
```

Figure 10.1.: Simplified ProtocolBehavior trait, in which generic-type parameters and method bodies have been elided for readability; the full version lives in the source repository [puffin] (protocol.rs).

```c
typedef struct AGENT_INTERFACE {
  // Lifecycle.
  void (*destroy)(AGENT);
  Res  (*reset)(AGENT, u8 name, u8 clear);

  // Streaming I/O and state progression.
  Res  (*add_inbound)(AGENT,
        const u8 *buf, usz len,
        usz *written);
  Res  (*take_outbound)(AGENT,
        u8 *buf, usz max_len,
        usz *read);
  Res  (*progress)(AGENT);

  // Claim callback: sole channel for
  // security-relevant internal events.
  void (*register_claimer)(AGENT,
        const CLAIMER_CB *cb);
} AGENT_INTERFACE;
```

Figure 10.2.: Simplified AGENT_INTERFACE C ABI, in which const qualifiers, header guards and logging helpers have been elided for readability (Res/u8/usz abbreviate RESULT/uint8_t/size_t); the full C header lives in the source repository [puffin] (puffin/puffin.h).

easing the addition of new PUTs, which mostly requires boilerplate code for the harness and the claim extraction: we harnessed LibreSSL this way in one day, demonstrating the low marginal cost of adding new PUTs.

Our DY fuzzers TLSPUFFIN is a crate that implements the protocol interface for TLS and notably specializes the protocol and properties description of TLS. It re-uses parts of the RUSTLS crate, that implements TLS in Rust, to build a *Mapper* for TLS and 240 of its operators (see Section 9.3.1). TLSPUFFIN also implements the *Objective Oracle* interface for TLS, which checks for agreement properties on session parameters. TLSPUFFIN is shipped with *Harnesses* for OpenSSL, LibreSSL, BORINGSSL, and WOLFSSL, which can be fuzzed out-of-the-box.

PUFFIN can be instantiated to other protocols, *e.g.,* SSHPUFFIN for SSH. That said, except when said otherwise, we focus on TLSPUFFIN and PUFFIN in the rest of the Chapter.

10.1.2. Implementation Challenges and Features

We review some challenges we tackled in building PUFFIN and TLSPUFFIN, which are detailed further in [*AHK23*, §E].

Performance Execution speed was key to our implementation choices. We chose to implement the communication between the agents through in-memory buffers, rather than relying on *e.g.,* TCP (which we offer as an additional feature though). We chose LibAFL for its high performance and parallel processing. Different fuzzer clients can be running on different CPU cores and share the same *Corpus*. For this, we had to make many components serializable: the term algebra and its link with $[\![\cdot]\!]$ (how to concretize), DY traces, claims, *etc.* See [*AHK23*, §C.1.4] for a performance evaluation.

Gathering *Knowledges* from *Claims* We observed that TLSPUFFIN often had to build some complex but public terms in traces, which correspond to hashes of the current transcript. Those dramatically increase the size and complexity of traces. We implemented a feature that, *w.l.o.g.* allow the attacker to invoke a routine to compute such sub-messages without having to provide a full attacker term for it. Furthermore, we stress that those two modifications do not change the executions and the attacker's behaviors that can be explored. We obtain the required data through additional *Claims* emitted by the PUT from which we extract specific *Knowledges*. For example for TLS and TLSPUFFIN, we extract the transcript hash from the PUT and make it available as a *Knowledge* for the attacker to use in traces (trough queries).

***Claim* extraction** PUFFIN offers different methods for extracting claims from PUTs. The preferred method is to leverage existing potential callback facilities in the PUT to expose the agent's context from which the required data can be retrieved. We used this method for all the current TLS harnesses. Another method we used for our first iteration with the harness of OpenSSL and LibreSSL (now legacy), is to create a minimal C interface of data the PUT must expose for PUFFIN to be able to extract the required data. We implemented this in a dedicated crate which can be used when the PUT API does not offer callback facilities. Finally, as

detailed in Section 9.3.2, some claims could also be extracted without any modification or even access to the PUT.

Additional Features It is possible to display (as graphically trees) and execute traces which are stored on-disk against any PUT, even against arbitrary TCP clients or servers (including remote and closed-source). We are already using our tools to test for regressions in the supported PUTs, treating existing attack traces as regression tests. TLSPUFFIN also offer features, notably its DSL, to allow developers to analyze TLS libraries with a similar goal of TLS-ATTACKER [Fit+20].

10.1.3. Implementing Differential DY Fuzzing in PUFFIN

We implemented the DDYF design from Section 9.4 in PUFFIN, which can be enabled through the CLI. DDYF is fully integrated into PUFFIN's three-layer modular architecture (Section 10.1.1). We evaluated it on TLS with OPENSSL, WOLFSSL, BORINGSSL, and LIBRESSL as PUTs (see Section 10.3) and also fuzzed WOLFSSH versus LIBSSH with SSHPUFFIN in differential mode.

At the **protocol-agnostic** layer (PUFFIN), we implemented the *Differential Oracle*, *Differential Harness*, and *Differential Observer*, all written once-and-for-all and reusable for any protocol. The *Differential Oracle* relies on the term algebra and a code annotation interface (see below) provided by the protocol layer. We also implemented CLI tooling (diff-execute) for triaging. This represents ca. 2.5k non-boilerplate LoC.

At the **protocol** layer (*e.g.*, TLSPUFFIN), the interface was extended with: the *Knowledge* code annotations (blacklist of fields to ignore and custom comparison functions), a set of common parameters for PUT configuration alignment, and the *decryption recipes* (Section 9.4.2). In contrast to DYF, no protocol-specific security policies need to be written. For TLS, this amounts to ca. 1k non-boilerplate additional LoC. At the **PUT** layer, DDYF requires ca. 200 additional LoC per PUT. Targeting other versions of an already-supported PUT requires only a dozen lines.

10.2. Evaluation of DY Fuzzing

We now present the results of the evaluation of TLSPUFFIN that we conducted in [*AHK24*], to which we refer the reader for the full methodology and additional details.

10.2.1. Finding DY Attacks

Our first goal is to answer the following **Research Question:** *How does TLSPUFFIN perform and compare with others at finding DY attacks?*

10.2.1.1. Methodology

Vulnerabilities benchmark suite To evaluate our work, we first establish a benchmark suite. The Magma project [HHP20] proposes a consolidated benchmark suite of 20 memory-safety related bugs in OPENSSL,

but none of these is a logical attack on which we could meaningfully evaluate DY fuzzing. A concrete example of such an out-of-scope vulnerability is the memory corruption in the ASN.1 encoder [CVE16-2108] which relies on a representation of zero as a negative integer. Such representation issues are out of scope of term-level modifications. Therefore, we selected 3 recent, known logical attacks (the 3 first vulnerabilities in Table 10.2) on OpenSSL and wolfSSL. To this initial ground-truth seed of known bugs, we add 5 newly discovered DY attacks.[1] The result is a first relevant benchmark suite of DY attacks, that we plan to augment in the future.

1: Note that tlspuffin found an additional new CVE after we finished this evaluation work [*AHK24*] (**CALLB** in Table 10.2), which raises the number of new vulnerabilities to 6.

Comparison with state-of-the-art fuzzers According to the following criteria we selected state-of-the-art fuzzers to be compared with tlspuffin: support for stateful PUT, tested on at least one cryptographic protocol, active project in the last 5 years. These criteria resulted in the selection of AFLNet [PBR20], StateAFL [Nat22], and AFLnwe [aflnwe], which were already plugged into the ProFuzzBench [NP21] benchmark tool.[2] All our experiments were conducted on a server with 500 GB of RAM and 2 AMD EPYC 7F521 processors with 16 physical cores each (but there are no strict minimum requirements for running tlspuffin).

2: One could argue AFLnwe was not designed for stateful PUT. We still consider it for completeness. It turned out it performed similarly to the others.

10.2.1.2. tlspuffin performance

tlspuffin is fast and can reach > 770 execs/s on a single core with LibreSSL as PUT. Fuzzing OpenSSL or wolfSSL is about half as fast. Enabling ASAN reduces performance by about 50%, which is to be expected [Ser+12]. About 84% of CPU time is spent in the PUT, and ca. 15% in trace mutations. tlspuffin scales and parallelizes well, as we observed that execs/s scales linearly with the number of available cores. The compared fuzzers reach <10 execs/s due to the overhead of creating a subprocess for each execution and sending messages through TCP.

10.2.1.3. Fuzzing Result Comparison

We ran each of the aforementioned fuzzers (with ASAN) three times for 24h fuzzing campaigns on wolfssl 5.3.0, which was affected by 6 vulnerabilities tlspuffin found; none of those 9 campaigns found any of them[3].

3: For being able to run those fuzzers and evaluate the achieved coverage (see Section 10.2.2), we had to resolve several issues and bugs in those projects (added support for wolfssl, and, more surprisingly, we found and reported 7 bugs in AFLnwe, StateAFL, and TLS-Anvil, including quite critical bugs).

Moreover, OpenSSL and wolfSSL are well-tested software and fuzzers are already continuously testing them. In particular, Google's large-scale oss-fuzz project [Ser17] continuously runs AFL++, honggfuzz, and libfuzzer [Fio+20; LLV; Goo] on OpenSSL as well as honggfuzz and libfuzzer on wolfSSL. The wolfSSL project itself runs 7 fuzzers internally every night (including a network fuzzer, libfuzzer, tlsfuzzer, and AFL) [Wol]. Those intensive fuzzing efforts were unable to find any of the 7 vulnerabilities from our benchmark suite.

In contrast, tlspuffin was able to find all 7 vulnerabilities. We ran 5 tlspuffin fuzzing campaigns on 12 cores that each found 5 of the 7 vulnerabilities within seconds or minutes. The detection of previously known CVEs SDOS1 and SIG takes more effort and is less reliable. We ran 90 fuzzing campaigns for 24h on 1 core each. For each of SIG and SDOS1, 5 of the 90 runs found the vulnerability with a mean time of 564min and 915min, respectively (we explain this higher variance in Section 10.2.2.2). The full methodology and results can be found in [*AHK24*, §C.1].

Result: *tlspuffin found 7 CVEs on OpenSSL and wolfSSL correspond-*

CVE ID	Alias	CVSS	Type	New	Version	TLS
2021-3449	**SDOS1**	5.9	Server DoS, M	✗	1.1.1j	1.2
2022-25638	**SIG**	6.5	Auth. Bypass, P	✗	5.1.0	1.3
2022-25640	**SKIP**	7.5	Auth. Bypass, P	✗	5.1.0	1.3
2022-38152	**SDOS2**	7.5	Server DoS, M	✓	5.4.0	1.3
2022-38153	**CDOS**	5.9	Client DoS, M	✓	5.3.0	1.2
2022-39173	**BUF**	7.5	Server DoS, M	✓	5.5.0	1.3
2022-42905	**HEAP**	9.1	Info. Leak, M	✓	5.5.0	1.3
2023-6936	**CALLB**	5.3	Info. Leak, M	✓	5.6.4	1.3

Table 10.2.: (Re)discovered vulnerabilities with TLSPUFFIN. The CVSS scores are the severity scores attributed by NIST. In the Type column, "P" indicates a protocol vulnerability and "M" a memory vulnerability (found by the DY attack). The "New" column indicates whether the vulnerability was first discovered using the TLSPUFFIN tool (✓) or rediscovered (✗). **SDOS1** affects OPENSSL. The others affect WOLFSSL. **Note:** We do not include here 3 other CVEs that TLSPUFFIN (re)discovered in differential fuzzing mode (DDYF); see Table 10.3.

ing to DY attacks, including 5 new ones which are reliably found in matters of minutes. Our selection of state-of-the-art bit-level fuzzers and others' fuzzing efforts found none of those 7 vulnerabilities.

In Section 10.2.1.5, we explain why the DY fuzzing approach is key and often necessary to find these vulnerabilities. We first review in Section 10.2.1.4 some of the (re)discovered bugs[4].

4: We provide the traces found by the tool that trigger those vulnerabilities and comprehensive vulnerability reports (for ours) in [puffin].

10.2.1.4. Summary of Vulnerability Descriptions

The **SDOS1** vulnerability allows malicious clients to crash OPENSSL servers during TLS 1.2 renegotiation by omitting the signature_algorithms extension but including a signature_algorithms_cert extension. **SIG** and **SKIP** are two bugs allowing client authentication bypass in WOLFSSL servers [RLD22] either by introducing a mismatch between the signature algorithm in the Certificate and CertificateVerify messages or by skipping the CertificateVerify message (both are encrypted and require to decrypt the server's response).

CDOS allows servers or MIM to crash WOLFSSL clients. It is triggered by sending 9 messages, ending with a large NewSessionTicket message (> 256 bytes) to a client with a non-empty session cache, who will then free a pointer to non-allocated memory and crash.

BUF is a stack buffer overflow bug in WOLFSSL servers. Malicious clients can cause a buffer overflow by sending specific Client Hello messages to servers: the list of cipher suites they offer must contain *duplicate* ciphers (at least 13); they should pretend to resume a previous session with the appropriate extensions and PSK; and they should omit the supported_groups extension. The trace from Example 9.3.2 can be mutated further to produce such an attacking trace. We explain in [*AHK23*, §C.3] why such a trace triggers the bug, its root causes, and how we proceeded to obtain such data. The triggerable stack buffer overflow has an attacker-controlled length with a maximum of 44700 bytes. Therefore, large portions of the stack can get overwritten, including return addresses. This vulnerability has the (unconfirmed) potential for Remote Code Execution (RCE).

HEAP is a heap buffer over-read bug on WOLFSSL servers using the WOLFSSL_CALLBACKS feature, meant for debugging but not discouraged for production. It is triggered by sending to a server a malicious Client Hello message with 25 extensions, notably a dozen key_share extensions. **CALLB** is similar but was found on a later version, after **HEAP** was fixed.

10.2.1.5. Advantages of DY Fuzzing

In Section 9.1.3, we explained why prior fuzzers are unsuitable for finding DY attacks; we now revisit these reasons in light of our results. *Why were all 7 vulnerabilities missed by the aforementioned campaigns despite intensive fuzzing efforts with state-of-the-art fuzzers?* We also explain why, on the contrary, DY Fuzzing is in a sweet spot for finding them.

(1) Reachability *(1.a) Harnesses* of most state-of-the-art bit-level fuzzers are often a limitation. We already mentioned the problem of many fuzzers [Fio+20; Zal16; LLV] that only send one flight of messages to the PUT, thus excluding all vulnerabilities we found, except **HEAP**. Some other fuzzers, including those we compared with as part of the ProFuzzBench suite, can send multiple flights but only act as clients to fuzz servers and are thus unable to find bugs in clients such as **CDOS**. In contrast, TLSPUFFIN can by design act as an attacking client, server, or as a MIM between PUT client and server. *(1.b) Mutations.* Some other vulnerabilities cannot be reached due to the limitations of the used mutations. Standard fuzzers, including those we compared with, rely on bit-level mutations that make it overwhelmingly unlikely that logical transformations of messages will be discovered. As an illustration, consider the attacker term $\mathsf{senc}(\mathsf{sdec}(x,k),k')$, which expresses that the attacker, instead of forwarding x, decrypts and re-encrypts x with a different key. The probability for this transformation to happen using random bit-level mutations is upper-bounded by the probability of breaking the encryption scheme. With DY Fuzzing, this adversarial behavior is obtained with a few mutations. All vulnerabilities except **HEAP**, **CDOS**, and **SKIP** require the attacker to apply such cryptographic computations. More generally, all of the 7 vulnerabilities rely on rather complex attacker terms obtained by a simple series of mutations (say t is mutated into t') such that the obtained bitstring $[\![t']\!]_\phi$ is very unlikely to be reached by bit-level mutations from $[\![t]\!]_\phi$. In practice, we empirically established that the fuzzers we compare with did not find **HEAP**, **BUF**, **SDOS1**, or **SDOS2** while TLSPUFFIN did, in the same testing environment: same seeds, harness, and detection capabilities (only ASAN).[5]

5: Doing the same for the others is impossible or challenging due to discussed limitations of the compared fuzzers (harness, detection capabilities).

Therefore, more-structured test-cases and mutations are needed, and the DY setting — which captures logical, adversarial behaviors — precisely provides a suitable model.

(2) Detection Classical fuzzers primarily aim at finding memory-related bugs, *e.g.,* with ASAN, but their *Objective Oracle* is unable to detect protocol vulnerabilities. In practice, even if a protocol vulnerability, such as the authentication bypass **SIG** and **SKIP**, *was reached*, the aforementioned bit-level fuzzers actually would simply drop this test-case as they would not realize that a policy violation occurred. (The 5 other vulnerabilities can be detected with ASAN.)

Again, a more structured approach is required, where a notion of session, agent (here channel), and claims are available to the oracle, as done in our DY Fuzzing approach.

On State-Machine-Guided Fuzzers We now compare with related structured fuzzers or testing engines from Section 9.1.3. **SIG** and **SKIP** were triggered by a state-machine learner [RLD22]; such learners are not fuzzers but a different approach capable of finding *some* DY attacks. They capture adversarial behaviors that drop or repeat whole Handshake mes-

sages without the ability to modify their content, except for a limited hard-coded pre-defined messages that can be used. In particular, [RLD22] is unable to reach **CDOS**, **SDOS2**, **BUF**, and **HEAP**. Moreover, such techniques do not feature an *Objective Oracle*; the output is a graph of execution flows that need to be manually interpreted to detect potential attacks. Similar conclusions apply to [Beu+15].

10.2.2. Coverage Comparison

Meaningfully evaluating fuzzers is notoriously difficult and code coverage is an unsuitable metric for this, despite being often used. This is backed up by Klees *et al.* [Kle+18]: "*Covering more code intuitively correlates with finding more bugs[...]. But the correlation may be weak, so directly measuring the number of bugs found is preferred.*" (cf. Section 10.2.1.)

That said, we shall cautiously compute and compare coverage to answer the following **Research Question:** *What specific insights about how* TLSPUFFIN *relates to state-of-the-art fuzzers can be gained by comparing code coverage?*

10.2.2.1. Methodology

Choice of fuzzers and testing engines We evaluated TLSPUFFIN in terms of coverage against our selection of bit-level fuzzers as well as the *combinatorial testing* suite TLS-Anvil [Mae+22]. Using ProFuzzBench, we gathered data about 24h fuzzing campaigns against wolfssl 5.3.0 and openssl 1.1.1j, that are affected by the vulnerabilities from Table 10.2.

Compute comparable coverage For the sake of fairness, when comparing different fuzzers, we only chose the seed test cases that can be expressed in all of the compared fuzzers, selected comparable Harness, and ensured line and branch counts are equivalent in all experiments. We also selected a subset of source files to be included in the coverage generation, excluding unit tests, fuzzers, examples and CLIs. We compute coverage over time using the gcov tool when executing the test cases of the corpus in the order they were found, which allows us to control the environment and the way the coverage is gathered, which is key to comparability across different fuzzers.

10.2.2.2. Code Coverage & Feedback Analysis

We first answer a question from Section 10.2.1.2 by studying TLSPUFFIN coverage: *Why did only 5% of experiments find **SIG** and **SDOS1**?*

To discover **SIG** and **SDOS1**, on average 3053 and 809 mutation tries were required, respectively (over 100 trials). (Note that the total number of executions in a fuzzing campaign can easily reach 10^8.) Therefore, those vulnerabilities are indeed reachable by TLSPUFFIN.

We also found that, once a certain diversity of DY traces is achieved in the *Corpus*, the current code-coverage feedback is "exhausted" and fails to promote DY traces that explore new PUT behaviors but only exercise code already explored (but in different states). To establish this, we analyzed the final corpora of the campaigns that did not discover **SIG** or **SDOS1**. We determined the total code coverage when executing all test cases in the corpus. We tested and observed that applying any

subset of the required mutations for finding **SIG** or **SDOS1** on traces from the final corpus did not improve this final code coverage. Therefore, such mutations, when applied one by one, will be discarded (as they are deemed uninteresting): the feedback metric is no longer able to measure progress. This happens when the *Corpus* becomes too large. It is however possible to find the required mutations before that point. Past that point, it remains possible (although very unlikely) to apply all required mutations at once. (Therefore, we recommend running multiple fuzzing campaigns versus only one for a longer period.)

Two key insights can be drawn from this: **Results:** *1. The code coverage is not an ideal feedback metric for DY fuzzings and needs to be improved (left as future work, see Chapter 11). 2. Code coverage is not a suitable metric to evaluate and compare DY fuzzers since finding new interesting DY traces will not immediately translate to better code coverage.* Another experiment supports this. We compared the achieved coverage when executing the trace triggering SDOS1 versus when executing the seed corpus only. The former covers a marginal amount of new code: no new functions were called (except error functions) and only +120LoC were covered (+0.1% of LoC).

10.2.2.3. Comparing Coverage of TLSPUFFIN vs. State-of-the-art Techniques

We first compared coverage achieved by TLSPUFFIN and TLS-Anvil, the best to-date test suite for TLS. A 24h fuzzing campaign on 32 cores with TLSPUFFIN achieves coverage on par or a bit less than what TLS-Anvil achieves (-1.6% of LoC were covered for WOLFSSL). We stress that TLS-Anvil contains hundreds of handwritten tests which are based on TLS-Attacker [Som16] and were designed to maximize coverage. *E.g.,* TLS-Anvil probes targets before testing then tries to do a handshake with every known cipher suite. This dramatically increases the coverage, because tested cipher suites are initialized on the server. Yet, TLS-Anvil found none of the 7 vulnerabilities.

We also compared the coverage of TLSPUFFIN against AFLnwe, AFLNet, and StateAFL. As previously mentioned, for the sake of fair comparison, we restricted TLSPUFFIN with the limitations of the former fuzzers plugged into ProFuzzBench: exclude clients from the *Harness* and use a smaller *Seed Corpus* (of 2 traces) by removing the 9 TLSPUFFIN seeds that were complex to produce for bit-level fuzzers (made easier for TLSPUFFIN thanks to our DSL). For all fuzzers and both targets, we executed 10 trials over 24h. For WOLFSSL, the mean branch coverage achieved by TLSPUFFIN across all trials is equal or greater than that of the compared fuzzers, and +33% when TLSPUFFIN uses all of its 11 seeds (see Section 9.3.3). For OPENSSL, TLSPUFFIN covers 6% less code, and 16% more code with all its seeds.

Manual inspection of the diff between the covered code shows that *bit-level fuzzers and TLSPUFFIN explore different parts of the code*. Bit-level fuzzers are good at exercising server-side code for handling features. A concrete example is cipher suites: since the signature and *Mapper* are currently not exhaustive (but could be made so with more work), TLSPUFFIN only implements 3 cipher suites out of 5 for TLS 1.3 and 3 out of 37 for TLS 1.2. Bit-level fuzzers discover them by flipping bits in the Client Hello. However, they are unable to later use such features (*e.g.,* a TLS 1.3 extension or a cipher suite) in subsequent messages as it usually requires logical message transformations. The code parts exercised this way can be huge (with a lot of setup and preprocessing work), even though it does not reflect a lot of different behaviors (since the features are actually not

used). As a concrete example, consider the file wolfssl/src/keys.c which essentially deals with cipher suites. AFLnwe covers 48.6% of the LoC in this file against 28.7% for TLSPUFFIN.

Showing the superiority of DY fuzzing at finding logical attacks by comparing coverage **Result:** *TLSPUFFIN explores fewer features but is better at actually using those features in subsequent messages.* We obtained empirical evidence supporting this claim by comparing the code exercised by TLSPUFFIN when executing detected vulnerabilities versus by long bit-level fuzzing campaigns. We focus on SDOS1 and BUF, as most other vulnerabilities exercise code that is not well harnessed by ProFuzzBench (*e.g.,* client-side code). The diff coverage between TLSPUFFIN executing those two vulnerabilities versus long bit-level fuzzing campaigns reveals key insights. *(i)* Features like secure renegotiation, required for SDOS1, need modifications under encryption, that were not reached by bit-level fuzzers but explored by TLSPUFFIN. *(ii)* BUF requires to produce a Client Hello message with fields computed by applying cryptographic primitives (decrypting application data, hashing, signature and HKDF). The code parsing this field is not exercised by bit-level fuzzers, which are unable to perform such logical transformations. (Our full coverage evaluation can be found in [*AHK24*, B.3].)

10.3. Evaluation of Differential DY Fuzzing

We now present the evaluation of PUFFIN's DDYF mode, described in Section 10.1.3. We answer the following research questions. **Research Questions:**

- *RQ1: Can DDYF detect vulnerabilities beyond the reach of DYF?*
- *RQ2: Are the results accurate and meaningful?*
- *RQ3: Is DDYF efficient?*
- *RQ4: Can DDYF fingerprint different PUT versions?*

We refer to [*GHK27*] for additional details including an ablation study of oracle components.

Benchmark: Detecting Known Vulnerabilities (RQ1) *Methodology.* We collected all 93 CVEs affecting PUT versions that PUFFIN fully supports: OPENSSL 3.1.2 to 3.4.0 and WOLFSSL 5.0.0 to 5.8.0. We filtered these to logical attacks within the scope of a DY attacker, excluding memory-only bugs (catchable by sanitizers), side channels, weak cryptography, and hardware-specific vulnerabilities. The resulting benchmark contains 5 CVEs affecting WOLFSSL: CVE-2022-25638 (certificate check bypass), CVE-2022-25640 (authentication bypass), CVE-2023-3724 (accepting an incorrect ServerHello leading to a weak encryption key), CVE-2023-6937 (bypassing encryption of server messages via a malformed coalesced flow), and CVE-2024-5814 (a downgrade attack in which a malicious TLS 1.2 server forces a TLS 1.3 client to accept a cipher suite it did not agree to). While this benchmark is small, it covers various types of logical attacks; its small size reflects an intrinsic bias in the CVE database towards memory-corruption bugs, for which mature fuzzing techniques already exist.

For each CVE, we checked whether (*i*) our *Differential Oracle* detects the vulnerability when given a manually written attack trace, and (*ii*) PUFFIN finds the vulnerability during a fuzzing campaign. For (*ii*), we ran 50 differential fuzzing campaigns of 5h each on 8 CPU cores between the

CVE ID	DDYF	DYF	% camp.	Time To Find Min.	Mean	Std.
2022-25638 (**SIG**)	✓	✓	66%	59s	2328s	2724s
2022-25640 (**SKIP**)	✓	✓	100%	<1s	38s	35s
2023-3724	✓	✗	100%	<1s	2s	1.2s
2023-6937	✓	✗	28%	420s	7130s	5151s
2024-5814	✓	✓	100%	<1s	8.5s	9.5s

Table 10.3.: Finding known CVEs with PUFFIN in DDYF mode. *Note:* **DDYF**/**DYF** indicate whether the CVE was found using the differential/standard oracle. **% camp.**: the proportion of campaigns where the CVE was found. Time statistics are over successful campaigns only (50 campaigns × 5h × 8 cores).

vulnerable WOLFSSL version and the latest supported OPENSSL version.

Results. Table 10.3 shows the results. All five CVEs were detected by the oracle and found during at least one fuzzing campaign. The first two and the last one (CVE-2022-25638, CVE-2022-25640, CVE-2024-5814) were already detectable by the pre-existing DYF security oracle; we confirm that our differential oracle also detects them. Recall that, in contrast to DYF, our oracle does not rely on manually written security policies.

The next two CVEs (CVE-2023-3724, CVE-2023-6937) could *not* be detected by DYF by design: they neither trigger memory corruption nor violate authentication properties checked by PUFFIN's DYF policies. CVE-2023-3724 causes a client to accept a ServerHello with a missing KeyShare and to use a weak encryption key —detecting such "secrecy" attacks in a DYF framework is very complex and was identified as challenging future work in [*AHK24*]. CVE-2023-6937 uses a malformed coalesced ServerHello to make a client accept unencrypted server data. Both were previously only discovered using symbolic execution [WA24], which relies on a manually-written TLS specification with limited configuration to check the PUT against. The fact that some CVEs are not systematically found could be attributed to the feedback exhaustion issue described in Section 10.2.2.2.

Finally, CVE-2024-5814 required extending the harness to support TLS 1.2 and TLS 1.3 simultaneously; with this extension, the downgrade attack is detected by both DYF and DDYF, with comparable success rates and time-to-find.

Result: *To the best of our knowledge, PUFFIN in DDYF mode is the first fuzzer able to find the two "secrecy"-style logical vulnerabilities CVE-2023-3724 and CVE-2023-6937.*

New Bugs and RFC Violations (RQ1,RQ2) *Methodology.* To check whether PUFFIN could find new bugs and vulnerabilities, we ran three 24h TLS differential fuzzing campaigns on 8 AMD EPYC 9275F CPU cores, testing OPENSSL 3.4.0 against WOLFSSL 5.8.0, LIBRESSL 4.2.1, and BORINGSSL (May 2026)*. We then triaged the resulting differences using our tool-supported triaging and analysis methodology (Section 9.4), resulting in 137 buckets against WOLFSSL, 84 against LIBRESSL, and 154 against BORINGSSL.

Results. Out of those buckets, 47 correspond to unique TLS RFC violations. Crucially, four are *Security-Related* bugs stemming from two defense-in-depth gaps that DDYF surfaced: a missing client-side defense against renegotiation attacks (affecting LIBRESSL, BORINGSSL, and WOLFSSL) and a missing point-at-infinity check in LIBRESSL's Elliptic Curve code. All prompted upstream fixes following our reports. Other RFC violations

* Using a single stack as the common baseline is a purely practical choice that limits the number of pairwise campaigns and the triaging effort. In principle, this is without loss of generality: two stacks that behave differently cannot both agree with the baseline.

include 8 *Wrong Accept/Reject* bugs (*e.g.,* 4 concerning HelloRetryRequest in wolfSSL, and LibreSSL accepting a TLS 1.3 cipher in TLS 1.2), 20 *Missing Alert* bugs (silent aborts), 14 *Wrong Alert* bugs, and 1 *Duplicate Alert*. These bugs are not necessarily security-critical but can lead to interoperability issues, make debugging more difficult, and show that all tested PUTs fail to strictly enforce the TLS state machine. Additionally, our experiments with SSH surfaced 5 RFC violations in two mature SSH stacks (wolfSSH and libssh). More details and a summary of all found bugs can be found in [*GHK27*, Table 2].

Result: *DDYF found 52 new RFC violations requiring complex adversarial message sequences, including four security-related (defense-in-depth) bugs, all since addressed upstream, but no critical zero-day vulnerability. By nature, such compliance bugs are difficult to find with non-differential techniques as they would require writing a specification of the expected behaviors. In contrast, DDYF is by design well-suited to find them.*

Accuracy and Performance (RQ3) To evaluate accuracy, we sorted the buckets of the campaigns above into two categories: (*i*) RFC violations or security-related bugs, and (*ii*) specification-allowed differences, *i.e.,* real differences due to different but compliant choices (*e.g.,* an OpenSSL server selects the first supported cipher of a ClientHello while wolfSSL selects its preferred one), including traces with several errors that PUTs parse in a different order, hence sending different Alerts. The proportion of buckets in category (*i*) is 3% for boringSSL, 12% for wolfSSL, and 26% for LibreSSL. The low ratio for boringSSL stems from several diverging choices unique to this stack and from its lack of a configuration API easing PUT alignment. To assess the robustness of our triaging, we ran 10 further 24h OpenSSL-vs-wolfSSL campaigns and applied the same triaging script: on average, only 0.05% of the traces (out of ca. 4.5M per campaign) remained untriaged, and 97.8% to 100% of the 137 buckets were filled. False positives are guarded against by manually reproducing and checking each reported bug against the RFC: all 52 were acknowledged by the vendors.

Regarding performance, the *Differential Oracle* introduces no measurable slowdown: the only (expected) overhead stems from executing each trace on two PUTs. An ablation study, disabling each *Differential Oracle* component in turn, shows that claims, status, and *Knowledge* comparisons all contribute (each is sometimes the only one to catch a difference), while most differences are caught by several components. The decryption component found no real difference in our TLS campaigns, but it did flag SSH traces exclusively, revealing RFC violations we would otherwise have missed. This is the first time a protocol fuzzer can meaningfully and structurally compare data under encryption.

Result: *Every Differential Oracle component contributes unique detections (decryption on SSH), at no measurable cost beyond executing two PUTs. The real bottleneck of DDYF thus remains the triaging process, whose cost is per bucket rather than per trace. Yet triaging scales: it leaves only 0.05% of traces unclassified, and every reported bug was acknowledged by the vendors.*

Fingerprinting Protocol Versions (RQ4) Given the high number of discovered behavioral differences, DDYF is well-suited to fingerprint different PUT versions. We compared puffin with an advanced active fingerprinting technique based on state-machine learning [RLD22] that found 2 clusters for wolfSSL versions 5.0.0 to 5.2.0. Using puffin in DDYF mode [*GHK26*, v1.0, Section 5.4.4], we quickly found two traces that dis-

tinguish *three* clusters: 5.0.0 alone, then 5.1.0 and 5.1.1 together (differing only by 3 LoC in memory management), and 5.2.0 alone—all completely stealthily through TCP without provoking a crash or memory violation. We plan to automate this methodology to systematically fingerprint PUT versions through the behavioral differences found by DDYF. We also plan to extend this analysis to more PUTs and versions, and to compare it with other fingerprinting techniques.

Result: *DDYF can produce appropriate traces to stealthily fingerprint protocol implementations, touching a sweet spot due to its ability to find a high number of diverse, non-crashing behavioral differences.*

10.4. Summary of Findings and Responsible Disclosure

In this Part, we presented the novel concept of DY fuzzing. We design a generic DY fuzzer, implement a DY fuzzer for TLS, and conduct a comprehensive evaluation thereof.

Using TLSPUFFIN, we discovered five new vulnerabilities in WOLFSSL. Each of them has been responsibly disclosed to and fixed by the maintainers of WOLFSSL. They have also been filed as CVEs as indicated in Table 10.2. For each of them, we offered our help, notably proposing fixes (some of them have been tested by TLSPUFFIN). Over the course of about 3 months WOLFSSL fixed all vulnerabilities with a mean time of 6 days until the fix landed on the main branch and a mean time until fixes were released of 26 days. A detailed timeline can be found below. Users of WOLFSSL like the OpenWRT[6] were informed by the publication of the CVE on WOLFSSL release days.

6: https://openwrt.org/advisory/2022-10-04-1

Later, in 2025 and 2026, we found 52 new RFC violations, including four security-related bugs, in the latest versions of WOLFSSL, OPENSSL, BORINGSSL, LIBRESSL, WOLFSSH, and LIBSSH using PUFFIN (TLSPUFFIN for TLS and SSHPUFFIN for SSH) in differential mode implementing our DDYF approach on TLS and SSH. These have also been responsibly disclosed to the respective maintainers. Since none of these bugs could be exploited (the security-related ones being defense-in-depth gaps), we did not file CVEs for them and disclosed the bugs through dedicated channels with the maintainers from September 2025 to October 2026. All the relevant stakeholders were very responsible: they have acknowledged and fixed most of these bugs in a timely manner.

August 12, 2022: Contacted WOLFSSL to set up a secure channel.
August 12, 2022: Reported CVE-2022-38153 and CVE-2022-38152.
August 12, 2022: Confirmed and fixed CVE-2022-38152.
August 16, 2022: Confirmed CVE-2022-38153.
August 17, 2022: Fix CVE-2022-38153.
August 30, 2022: Release of a fix version 5.5.0.
September 12, 2022: Reported CVE-2022-39173.
September 12, 2022: Confirmed and fixed CVE-2022-39173.
September 28, 2022: Release of a fix version 5.5.1.
October 9, 2022: Reported CVE-2022-42905.
October 10, 2022: Confirmed and fixed CVE-2022-42905.
October 28, 2022: Release of a fix version 5.5.2.
October 31, 2023: Reported CVE-2023-6936.
November 6, 2023: Confirmed and fixed CVE-2023-6936.
October 28, 2023: Release of a fix version 5.6.6.

September 2025–October 2026: Disclosure of and fixes for the 52 new RFC violations (including four security-related bugs) found by PUFFIN in differential mode (DDYF) on WOLFSSL, OPENSSL, BORINGSSL, LIBRESSL, WOLFSSH, and LIBSSH.

11. General Perspectives

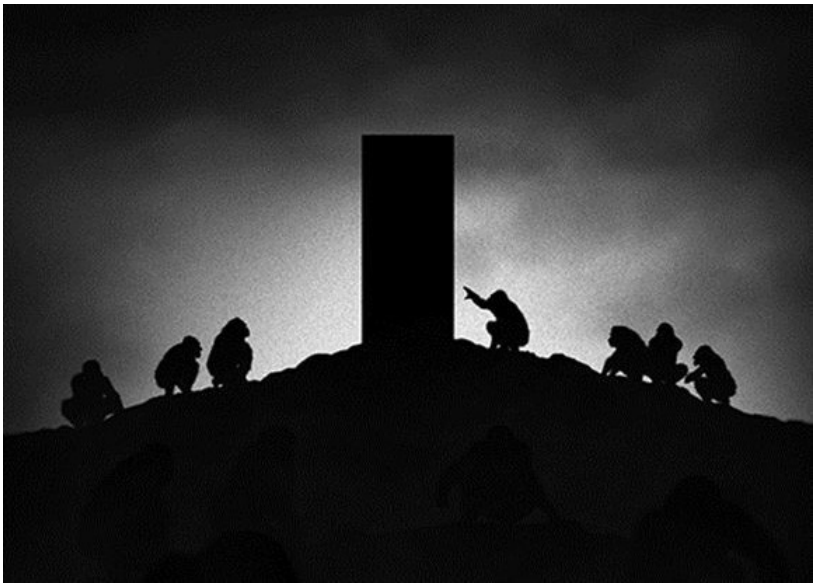


Figure 11.1.: Mysteries of the future. The "Dawn of Man" sequence from *2001: A Space Odyssey* (1968). Directed by Stanley Kubrick. © Warner Bros. Entertainment Inc. Used for educational and academic purposes under the short-citation exception (or "fair use").

11.1 Formal Verification Perspectives 187
11.1.1 Unified Lattice Framework 187
11.1.2 Composite Arguments and Lattices 188
11.1.3 Optimizing Verification Using Earlier Results . . 189
11.2 E-Voting Protocol Perspectives 190
11.3 DY Fuzzing Perspectives 191
11.3.1 Improve the DY Fuzzer Feedback 191
11.3.2 Broaden the Scope of the DY *Objective Oracle* . . . 192
11.3.3 Semi-automating *Mapper* Construction for New Protocols 193
11.3.4 Combining DY and Bit-Level Fuzzing 194
11.3.5 Advancing Differential DY Fuzzing 195
11.3.6 Engineering Infrastructure and Longer-Term Maintenance 195
11.4 Bridging the Gap Between DY Fuzzing and Verification 196
11.4.1 Hybrid Framework and Fuzzing 196
11.4.2 Model Extraction and Refinement 197

Looking back at the contributions gathered in this manuscript, a coherent program emerges: applying and advancing formal methods to study, attack, and strengthen real-world cryptographic protocols, with formal verification and DY fuzzing as two complementary techniques both grounded in symbolic models. Parts A and B extended the reach of symbolic verification along several axes —richer equational theories, large-scale industrial protocols, fine-grained threat-model exploration, and effective privacy analysis. The e-voting analyses (Part B) anchored both strands in concrete societal stakes, while Part C proposed DY fuzzing as a new bridge between symbolic models and protocol implementations. Several open questions persist, and others only became visible once these foundations were in place; together, they shape the agenda for the coming years.

We begin with perspectives in formal verification, then turn to avenues for future work on e-voting protocols, before discussing perspectives for DY fuzzing and, finally, its interplay with verification. Before that, we briefly comment on LLMs.

A Personal Note on LLMs The first draft of this manuscript was written in early 2026, when it was not yet completely obvious to us that the rise of LLMs would dramatically collide with formal verification and software testing beyond helping with writing boilerplate code. By fall 2026, things have evolved quickly, and one cannot ignore how much this technology is already changing our field.

The enormous promises made by LLM promoters are on par with the negative externalities (*e.g.,* CO_2 emissions) this technology also has; these must not be overlooked. New frontier models are released every few months, in a seemingly never-ending cycle of ever more capable models.

In theory, this cycle could soon converge to good-enough, open-weight models that run locally and no longer need to be retrained, which would mitigate these externalities. Yet nothing indicates that development is heading toward such a fixed point. We do not plan to resolve this debate within the scope of this manuscript.

However, as researchers, we constantly question our research directions, and this was yet again an occasion to reflect on our priorities and to ask whether our research agenda would still be relevant from 2027 onwards. We believe it will be.

Formal verification will continue to play a central role in ensuring the correctness and security of cryptographic protocols. LLMs are unreliable and their output cannot be trusted as such, which is why the recent trend of training them to produce machine-checked proofs is gaining so much traction: every proof they propose is then checked by a sound tool. Yet this heavily relies on a solid foundational understanding and on rigorous formal methods: a machine-checked proof is only as meaningful as the model and the security properties it is about, and writing these remains the hard part. We have recently seen many ungrounded or even delusional attempts to fully close the automated verification loop without a deep understanding of the underlying principles.

We believe fuzz testing methods will change even more. In stark contrast to formal verification, these methods are stochastic in nature, very much like LLMs, whose capabilities at finding software vulnerabilities have improved enormously in recent months and will likely continue to do so in the near future. With every new model iteration, stakeholders who can afford the token costs will analyze cryptographic protocol implementations with the latest, most capable LLMs, which may uncover previously unknown vulnerabilities and weaknesses. However, (fuzz) testing still has a lot to offer.

First, although it does not provide formal guarantees, it tells us what was explored: coverage can be measured (even if code coverage *alone* is a poor proxy for finding bugs, see Section 10.2.2), and for DY fuzzing the search space is defined by an explicit attacker model. A negative result thus has a clear meaning (no attack found within this attacker model and this budget), which is not the case when an LLM reports that it found nothing. In this sense, although fuzzing is also a probabilistic method, it is more principled and offers informal positive assurance when it finds no bug[1].

1: This assurance can even be quantified statistically, *e.g.*, by estimating the residual risk of stopping a campaign [Böh18; BLW21].

Second, it can be greatly enhanced by combining it with LLMs throughout the fuzzing process: (i) LLMs can assist with all the boilerplate code that is necessary to fuzz deeper and wider (*e.g.,* with agentic LLM assistance, we harnessed LibreSSL in one day, see Section 10.1.1); (ii) LLMs can assist with triaging: fuzzing could take the role of producing large volumes of bug candidates that LLMs could then triage, combine, and try to exploit (we started exploring the triaging part with DDYF; see [*GHK27*, §4.3 and §5.4.3] and Section 11.3). In both cases, fuzzing grounds the work of LLMs in concrete, reproducible executions, which, we believe, makes the combination more reliable than LLM-only approaches. This is also the use of LLMs we find reasonable given the externalities discussed above: as helpers, for which smaller, locally run models are often enough, while the evidence comes from the fuzzer, its oracles, and the formal models.

We would answer a devil's advocate arguing that LLMs will become so effective at directly finding bugs that fuzzing will only play a marginal role: even then, LLM-based agents will need executable ground truth

to confirm their findings, and diversity of approaches is valuable, since agents driving fuzzers and formal models as tools (rather than replacing them) could tackle areas and scenarios that would otherwise be missed. An early illustration was DARPA's AI Cyber Challenge in 2025, whose finalist systems combined LLMs with fuzzing [Kim+25]. We therefore still believe in principled testing methodologies, possibly with LLMs in the loop.

That said, the technology is advancing so rapidly that it is challenging to predict its future trajectory; one must remain cautious and continuously reassess the implications for both research and society. Beyond the immediate research challenges, our other missions are more central than ever, notably passing on knowledge, through the education of future researchers and students, and through its dissemination to the wider public when relevant.

11.1. Formal Verification Perspectives

The contributions presented in Parts A and B mainly raise three families of open problems, concerning:

- the generalization of our lattice-based methodology to provide fine-grained analysis results,
- the development of composite arguments for improving the scalability and scope of symbolic verification, and
- the exploitation of earlier verification results across the joint lattice of properties and threat models, to optimize both individual proof effort and the overall verification effort.

Building on these three directions, we will leverage the lattice framework to develop a novel verification methodology that encompasses all three. This way, we will improve not only the depth of the analyses but also the scalability and scope of symbolic verification in general.

Beyond these, we also plan to continue identifying gaps in the modeling of cryptographic primitives and protocols, in the spirit of recent, more precise models for hash functions (Chapter 3), Exclusive OR (XOR) (Section 4.1.1), signatures [Jac+19], Diffie-Hellman [CJ19], KEM [CJL23], hybrid constructions [Bha+24a; Laf+25; LMR24], *etc.* As we have always done, we will also continue to apply our verification techniques to real-world protocols and deployments, in order to both validate our techniques and to provide useful insights and results for practitioners. This is also how we plan to identify new modeling and verification challenges to be tackled in the future.

11.1.1. Unified Lattice Framework

The framework and tooling for the different lattice explorations we developed in [*Gir+20*; *Che+23*; *DHK25a*] and presented in Chapter 5 differ across studies, and we initially preferred adapting to the specificities of each case study over devising a generic framework and exploration script. However, now that we have more perspective on this approach, it would be worthwhile to design and implement a modular, generic lattice-of-proof-obligations framework. We paved the way in [*DHK25a*] with a modular lattice library, but more work in this direction is needed.

Our vision is to replace the current manual, single-objective verification process, in which the analyst must explicitly specify —in a formal language— both the properties to check and the system and threat as-

sumptions under which to check them. The challenge is to automatically *discover* which security and privacy properties hold and under which threat models and potential protocol and system assumptions.

The first challenge is to *automatically identify the relevant protocol-dependent combined lattice of threat assumptions and properties* with subsumption (*i.e.,* implication) as an order. The satisfied maximal points in this lattice —where the property holds under the given assumptions— will then be computed by our verification framework. The second challenge is to propose *new formal definitions and models* in symbolic logic for emerging security and privacy challenges. The third challenge is to *represent the results of such analyses to the user in practitioners' terms*: specifically, the *security impacts*, the underlying *trust and threat assumptions*, and the inherent *trade-offs* between stronger threat models and stronger guarantees. This way, we see formal verification as an ideal tool to *measure*[2] the security guarantees provided by a protocol and to *precisely qualify its privacy loss*.[3]

Concretely, such a unified framework would allow us to easily define and explore features and optimizations across different protocols and properties, ease evaluation and reuse across case studies, and be readily extended to new protocols and properties. It will also serve as a testbed for the novel verification methodology we plan to develop, as explained in the next sections.

2: Other aspects relevant to risk analysis, such as the likelihood of certain threats, are however outside the scope of our framework.

3: In particular, we believe this could be a valid answer to the challenges we raised concerning the formal analysis of privacy in mobile communication protocols; see Section 7.4.3.

11.1.2. Composite Arguments and Lattices

Using automated provers to *measure* protocol security also offers an opportunity to identify non-trivial relations between verification goals. In a sense, our line of work on sufficient conditions for privacy [*HBD16*; *HBD19*; *HC19*] discussed in Section 4.2.2 can be recast through the lens of this lattice framework.

Stepping back, this methodology intuitively consists of devising several conditions such that: (i) all conditions together imply the complex property under consideration (*soundness*), (ii) each condition is fundamentally simpler than the original property and can be efficiently checked —more efficiently and with better precision— (*checkability*), (iii) the conditions are met by real-world protocols and false attacks are minimal (*tightness*). The crux lies in finding sub-conditions satisfying (ii) and (iii), thereby enabling verification without systematic false attacks. For instance, it was of great interest to prove an implication of the following form [*HBD16*; *HBD19*] (where $\mathsf{M} \vDash P$ denotes that the property P holds in the symbolic model M):

$$\underbrace{\mathsf{M} \vDash \mathsf{StrongAuth.}}_{\text{reachability property}} \wedge \underbrace{\mathsf{M} \vDash \mathsf{DataPrivacy}}_{\text{message equivalence}} \Rightarrow \underbrace{\mathsf{M} \vDash \mathsf{Unlinkability}}_{\text{equivalence property}}.$$

Indeed, state-of-the-art verifiers usually have no problem verifying reachability properties or equivalence properties over messages[4], in contrast to full equivalence properties (see Sections 4.2.2 and 7.4.3). This methodology allows the proof to be modularized as well. More specifically, it is often possible to check some conditions, often reachability properties, that capture large classes of attacks based on control-flow issues and executions that deviate from the expected, honest one. One can then focus on the simpler sub-problem restricted to executions satisfying those conditions —that is, executions that do not deviate too far from the honest execution— setting aside the many other possible behaviors. This allows

4: Such equivalence properties, akin to the notion of *static equivalence* [AF01], are simpler to verify than full trace equivalence since they apply only to the resulting sets of messages rather than to the entire execution flow. Consequently, they are less affected by the approximations typically required by automated verifiers. In particular, one can effectively leverage *diff-equivalence* to capture such properties [*HBD19*].

for a gradual simplification of the verification problem.

It would be interesting both to identify further *composite arguments* within current lattices and to extend those lattices with properties that are individually easier to prove and that create new opportunities for such arguments.

To do so, one could explore four levels of properties (i) to (iv). At level (i) are simple invariants local to a single agent, such as secrecy and freshness of nonces and keys. At level (ii) are global invariants about message flow and control flow, independent of message content, such as agreement properties. Establishing such properties first enables the use of control-flow simplifications as previously explained and formally established in [*HBD16*; *HBD19*; *HC19*]. At level (iii) are global invariants about message contents, including strong secrecy (modeled as a non-interference property) and other message-indistinguishability properties —more generally: what data relations are hidden *from* the attacker? Finally, at level (iv) are complex equivalence properties such as unlinkability. Classical secrecy and authentication properties are directly derivable from (i) and (ii). Precise privacy guarantees (notably (iv)) and privacy loss can be deduced from (ii) and (iii): what does the protocol leak to the adversary in terms of relations between data, agents, and actions? More generally, from (i)–(iii), indistinguishability between protocol and idealized protocol can be deduced.

11.1.3. Optimizing Verification Using Earlier Results

We discussed how to adjust the order of verification goals fed to the verifier (the so-called *exploration strategies*, Section 5.4), but this does not allow the single-objective verification process to benefit from the results of prior goal verification.

First, carefully selected established results can serve as axioms to speed up subsequent verification. One could also exploit the intermediate information produced internally while verifying earlier goals to dynamically detect further subsumption, thereby eliminating redundant queries. We plan to explore these directions in the context of our unified lattice framework, which will allow us to easily define and evaluate different optimizations and strategies for leveraging prior results in the verification of new goals. We provide next two examples of such strategies.

Example: incomparable threat models and properties For instance, one could devise and prove non-trivial relations between properties under non-uniform threat models, exploiting such information. For example, if a threat model t_i equals t plus a specific intruder capability c_i, we can use the fact that a property P has been analyzed under t_1 when checking for P under some other threat model t_2 by:

1. Analyzing the potential attack trace under t_1: does it use c_1? If not, we can *statically* conclude that P does not hold under t_2 either.
2. If P holds under t_1, then any attack on P under t_2 must exploit c_2 —the only new capability available— and one can leverage this restriction to optimize the single-objective verification of P under t_2.

It will also be interesting to optimize proof search to maximize such inter-objective interactions —for instance, by prioritizing branches in which the attacker exploits the "weakest" capability first. We believe these proof strategies could be implemented through selection function tweaking for ProVerif (see Section 2.4.2) or exploration oracles and tactics for

Tamarin (see Section 2.4.1.3). Another interesting direction is to study the impact of such objective interactions on the aforementioned lattice exploration strategy.

Example: proof-helper methodology for ProVerif Closely related is the ProVerif proof methodology we developed for OPC UA (Section 5.6), based on chains of noselect statements (see Section 2.4.2) and iteratively added invariants. Although we designed this methodology solely to manage the complexity of OPC UA, deriving invariants that are specific to that protocol and not of independent security interest, it is worth revisiting through the lens of the lattice framework to systematize it and provide better tool support.

As currently implemented, the methodology suffers from two fundamental limitations. First, the methodology is highly *experimental and fragile*: noselect statements are identified by manually inspecting ProVerif's verbose output, and invariants are added empirically by analyzing each cannot-be-proved clause in turn. Second, even when the invariants are available, they are applied *opportunistically* by ProVerif (without unification forcing), meaning the proof is not truly *guided* by them but only lazily assisted, which makes the outcome brittle.

We plan to redesign this methodology along two lines. First, rather than accumulating invariants empirically, we will construct an explicit, structured *proof scheme* in the style of a proof assistant, where each step could discharge a sub-goal automatically and where the proof obligations reflect the protocol's known structural invariants (for OPC UA: channel authentication is established first, from which session-table entries follow, then session activation). Second, the established invariants should actively *guide and restrict* the proof search to executions consistent with them rather than being applied lazily by the resolution engine. This requires rethinking parts of the resolution algorithm. We expect this to yield proofs that are more modular, more robust, and more systematic in their construction, and we plan to accompany this with dedicated tool support, *e.g.,* for managing the dependency graph between invariants and verifying that no circular reasoning is introduced[5].

5: Assuming invariant I_1 is needed to prove invariant I_2, then I_2 should not be used to prove I_1.

11.2. E-Voting Protocol Perspectives

Building on our expertise in real-world electronic voting protocols and deployments (Sections 8.5 and 8.6), we plan to continue proposing practical and formally proven solutions to bridge the gap between current state-of-the-art academic protocols and practical constraints arising from real-world deployments.

First, we have shown (Section 8.5) that some lawful requirements governing when a quorum is met to *e.g.,* decrypt a ballot-box are not cryptographically enforced during French legislative elections. As a consequence, fewer people than legally prescribed can collude and decrypt a ballot-box. This also affects other protocols and national elections worldwide. We shall design a solution to cryptographically enforce such operational constraints to prevent any human misbehavior.

Second, vote-tagging is a mechanism that tags each ballot with its voter's attributes (*e.g.,* age range). This is used for statistical purposes or to create other *indirect elections* within a smaller scope (*e.g.,* eligible voters over fifty years old). It is widely used in practice, especially during

professional elections. However, the widely-used solutions today (such as encrypting the voter's choice and their attributes in the same message) are clearly suboptimal and disclose much more information than necessary, negatively impacting vote secrecy. Moreover, the verifiability guarantees offered for the direct election do not carry over at all to the indirect elections. We would like to propose solutions that limit mission creep while ensuring guarantees for vote secrecy and verifiability for both direct and indirect elections.

Third, we note that almost all proposed e-voting protocols rely on classical cryptography, *i.e.,* cryptographic primitives whose security relies on hard problems that do not resist quantum computers (see Section 1.2.1). Therefore, a transition towards post-quantum cryptography has begun (see *e.g.,* [BHM21]). It consists of using cryptographic primitives whose security relies on hard problems expected to remain difficult when considering quantum computers. However, this brings new threats as post-quantum cryptography is less well-understood and scrutinized, and raises the question of the real difficulty of these new problems. To temper this risk, the design of *hybrid-secure protocols* is an appealing approach: they combine both classical and post-quantum cryptography in a way that their security holds as soon as one of the two underlying hard problems remains difficult. We plan to design a hybrid-secure e-voting protocol by building upon secret-sharing and general-purpose zero-knowledge proofs (GPZKPs), in the spirit of [BHM21] (a direction we started exploring with A. Outrey during his internship and are now pursuing in his PhD).

Finally, we plan to explore the application of fuzzing to e-voting protocols, which has surprisingly not been explored, neither in academia nor by practitioners (B. Voisin began investigating this with us during his internship and continues in his PhD, targeting the Swiss Post e-voting system as the first case study). There are several challenges to adapting DY fuzzing techniques to e-voting: (i) they involve complex workflows and interactions between many different actors, (ii) protocols are expected to resist strong threat models, including insider threats and collusions between different actors, calling for designing dynamic actor compromise within the DY fuzzer, (iii) security properties are more complex and require designing new *objective oracles* capable of detecting violations thereof.

11.3. DY Fuzzing Perspectives

DY Fuzzing instantiates a new paradigm —fuzzing guided by symbolic, formal protocol models— and opens a broad research agenda that we now detail. We first discuss various directions for future work to improve and extend DY fuzzing and PUFFIN in its current form. Several of these directions share a common thread: better exploiting the underlying DY model, which today drives input structure and DY security policies but whose semantic richness remains only partially exploited by the fuzzing loop itself. We will then discuss in Section 11.4 perspectives for the interplay between DY fuzzing and formal verification.

11.3.1. Improve the DY Fuzzer Feedback

We established in Section 10.2.2.2 that coverage-based feedback gets exhausted and does not promote semantic diversity (in the sense of the DY model). It is the feedback space, not the fuzzing space, that saturates first[6]. By analogy with *overfitting* in machine learning, PUFFIN over-specializes

6: As an illustration, consider a TLS authentication bypass triggered by a specific combination of TLS extensions. Each individual extension may already have been exercised by traces in the *Corpus*, and a full handshake may also have been completed from the outset, since some *seeds* already do so. In such a case, the specific combination of extensions required to trigger the bypass may fail to be promoted, as it yields no new code coverage, and this exploration path will therefore never be pursued: the code coverage has been exhausted.

toward maximizing code coverage, ignoring test cases that are semantically novel but cover no new edges; retaining some such cases at random —in the spirit of *dropout regularization*— could prevent premature corpus convergence.

The deeper fix is to design *DY-specific feedback metrics*, such as coverage of the DY message space and protocol state space, diversity of generated attack traces, or proximity to known attack traces. A concrete starting point is tracking ordered sequences of cryptographic operators and message constructors along subterm paths of accepted adversarial terms —a form of structural diversity that code coverage cannot capture. More generally, protocol *features* or *feature combinations* could be tracked using an underlying feature grammar, in the spirit of grammar-based coverage. DY verifiers could further estimate each test case's *proximity to a known attack trace*, providing a principled progress signal and also guiding our generative mutations —currently random and blind— by synthesizing targeted terms.

Finally, annotation-driven fuzzing (notably IJON [Asc+20] and SGFuzz [Ba+22]) has shown that domain-specific annotations exposing internal program states can steer a fuzzer toward semantically interesting executions without relying solely on code coverage; we plan to bring this idea to the DY level by annotating the PUT with DY-specific information —such as message types, protocol states, and exposed agent internals— to guide the search toward meaningful adversarial behaviors and explore semantically diverse protocol executions.

A natural way to operationalize DY-specific feedback is to extend the classical code-coverage signal into a small *tuple* of orthogonal dimensions: classical code coverage; the *initial state* of the trace (which agents, in which configuration, so that hits coming from *e.g.,* a MIM adversary are not conflated with those from a malicious client); a *trace-shape* signature, *e.g.,* a hash of the input/output skeleton; and a bounded fingerprint of the *adversarial terms* exercised so far (as discussed above), as a finite proxy for semantic diversity. A central design tension is to avoid combinatorial explosion[7]: enriching the feedback with too many dimensions may end up promoting too many traces and fail to prioritize the most promising ones, while too few dimensions may fail to promote semantic diversity.

7: We should distinguish between coverage metrics used for fuzzing (feedback) and for evaluation (for comparison purposes). The former needs to be bounded and selective, but not necessarily the latter.

11.3.2. Broaden the Scope of the DY *Objective Oracle*

Currently, the TLSPUFFIN *Objective Oracle* captures memory-safety bugs and authentication violations through DY security policies. However, classical DY verification of specifications operates on a richer class of properties. For instance, secrecy properties are expressed by the attacker's inability to *deduce* a term corresponding to allegedly confidential data.

To reason about attacker deduction, we would first need to *fully reinterpret* the bitstrings obtained by output actions as terms: a total parsing function $\lfloor \cdot \rfloor : \mathsf{M} \rightharpoonup \mathcal{T}$ that computes the inverse of $[\![\cdot]\!]$ for *all terms*, see Remark 9.2.3 and Section 9.3.1. Doing so will likely require additional claims from the PUT about the term structure of the sent bitstrings (in the spirit of [MK24]; see the hash function example in Remark 9.2.3). We would then leverage standard decision procedures for deduction (*e.g.,* [AC06]). Achieving this *reinterpretation* seems challenging but would be crucial to tighten the connection between DY fuzzing and verification, and to extend the scope of DY fuzzing to a much broader class of properties, including confidentiality properties. A further relevant property to capture is *functional correctness* with respect to the DY model of the underlying protocol.

Finally, we plan to extend the *Objective Oracle* to capture privacy properties such as anonymity and unlinkability, which are expressed as *equivalence* properties in the DY model (see Section 2.3.3). Here, our differential DY fuzzer (DDYF, Section 9.4) offers a natural starting point: comparing two executions for output indistinguishability can serve as a concrete operational approximation of such equivalence-based objectives.

Beyond these *property* extensions, we also plan to enrich the *threat-model scenarios* covered by the *Objective Oracle*. Classical DY analyses routinely consider dynamic adversary capabilities such as long-term key compromise, session-key reveal, and partial corruption of subsets of parties; reflecting these in PUFFIN requires extending its base agent-interaction scenarios and exposing the corresponding compromise actions to the mutators. More fundamentally, this calls for *reconciling PUFFIN's operational threat model* —what attacker capabilities the fuzzer actually offers— with the formal DY threat model assumed by downstream verification.

11.3.3. Semi-automating *Mapper* Construction for New Protocols

Onboarding a new protocol to a DY fuzzer is currently a substantial engineering effort and one must in particular write a protocol-specific *Mapper*: enumerate the protocol's operators and link these to executable functions, write the (de)serialization layer mapping symbolic terms to bitstrings (*i.e.,* $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$). This work is currently largely manual, and it remains the single largest barrier to applying DY fuzzing to new targets. It also raises a more fundamental concern: how can we ensure the resulting *Mapper* is *exhaustive*, *i.e.,* that the fuzzer can in principle construct every term of interest in the protocol's term algebra?

Our vision is to *semi-automate Mapper* construction by extracting the symbolic signature directly from the protocol implementation's type structure: for each message type, a constructor function symbol is generated with arity matching the type's fields, and projection (deconstructor) symbols are derived in turn; the programmer provides only a thin layer of annotations identifying cryptographic primitives and the fields to expose to the attacker.[8] The engineering cost of integrating a new protocol collapses to a few targeted annotations, and the extracted signature systematically covers the protocol's term algebra rather than whatever subset a developer happened to encode —providing a principled notion of *exhaustiveness* and aligning our *Mapper*s with the *constructor/deconstructor* discipline of symbolic models such as PROVERIF and TAMARIN.

8: We are exploring this direction as part of T. Gouville's ongoing PhD.

Several research questions arise.

- *Statefulness:* real-world implementations manipulate persistent state —transcript accumulators, session caches, sequence counters— that does not naturally lift to pure function symbols; bridging this gap likely requires new abstractions where state itself becomes a first-class term.
- *Subterm sharing and trace representation:* a related design opportunity is to represent traces as *directed acyclic graphs (DAGs)* rather than trees, with shared subterms evaluated once and mutated coherently. This would simultaneously address evaluation cost, mutation coherence (*e.g.,* keeping a cipher-suite choice consistent across messages), and scalability to large traces, but it calls for new mutations together with guidance rules preventing degenerate constructor/deconstructor chains, in the spirit of TAMARIN's normal-form

restrictions (see Section 2.4.1.2).

- *Automated seed synthesis:* DY fuzzing campaigns currently start from manually written seed traces, which becomes an obstacle as *Mapper*s grow. We plan to automate this step: from a captured benign protocol exchange, the bitstrings can be reinterpreted as terms and lifted into a DY seed trace, drastically lowering the cost of bootstrapping a campaign on a new protocol.
- *Quantifying exhaustiveness:* an exhaustive *Mapper* also raises the question whether the DY attacker operating the fuzzer can in principle reach all terms of interest in the protocol's term algebra. Dedicated coverage notions over the term algebra appear promising and tie this direction back to the DY-specific feedback metrics discussed earlier.

We plan to validate this approach by migrating the existing TLSPUFFIN's *Mapper* to the automated framework —comparing size before and after as a quantitative measure of the engineering benefit— and by onboarding new protocols such as Signal, SSH, OPC UA, and WIREGUARD. Combined with the model extraction discussed next (Section 11.4), semi-automated *Mapper* construction would close the loop between DY models and protocol implementations, letting a single annotation effort drive both fuzzing and symbolic verification.

11.3.4. Combining DY and Bit-Level Fuzzing

Structured fuzzing —of which DY fuzzing is an instance— and classical bit-level fuzzing pursue opposite goals. Bit-level fuzzers apply unstructured mutations (bit flips, byte increments) to deliberately escape the input grammar and expose parsing bugs, malformed headers, and invalid length prefixes. Structured fuzzers, by contrast, work modulo a grammar or term algebra to reach deep protocol states, but cannot operate on raw bitstrings: re-parsing a bit-mutated message defeats the purpose, since parsing fails precisely when the mutation succeeds in escaping the grammar. The two approaches thus appear irreconcilable, yet each in isolation is strictly less expressive than their combination.

We aim to reconcile both within a single framework. The central idea is a notion of input that *retains term structure* while allowing any subterm to be *frozen as the bitstring produced by its evaluation* and subsequently bit-mutated. This raises a non-trivial algorithmic challenge: when the surrounding term is concretized into a final bitstring, the frozen bitstring —which may occur several times or even be empty— must be *located and replaced in place* rather than re-parsed.[9]

9: Indeed, parsing would certainly fail once bit-level mutations are applied on the frozen bitstring.

The resulting fuzzer goes far beyond a naive "DY-mutate first, then bit-level-mutate" composition: bit-level mutations apply at any depth, including under complex formatting (length-prefixed fields, magic constants) and under opaque cryptographic operations (encryption, signature). Moreover, later messages in a trace *adapt automatically* to earlier mutations, since they are computed over the PUT's outputs, which themselves shift accordingly. This also opens the door to *new structured transformations* unreachable by DY mutations alone, by bit-mutating a subterm and re-interpreting the resulting bitstring as a different —and possibly malformed— structured message.

Beyond bug-finding, the unified framework will finally enable *fair, controlled comparisons* between paradigms —DY-only, bit-only, and combined— on *exactly* the same harness and PUTs, which has so far been impossible. We plan to systematically evaluate the gains in coverage and vulnerabil-

ity discovery, and to conduct ablation studies on the key design choices (in-place bitstring replacement, mutations under opaque symbols, adaptive trace continuation, *etc.*).

11.3.5. Advancing Differential DY Fuzzing

Our differential DY fuzzer proves effective at uncovering bugs, but it remains in its infancy and raises many research questions we plan to explore.

- The main limitation of DDYF is the human effort required to triage the large number of flagged objectives. We plan to design an automated triaging pipeline that groups differences into clusters, splits them into finer buckets each corresponding to a single root cause, and ranks buckets by potential impact. A complementary avenue, which we started exploring in [*GHK27*], is LLM-assisted summarization and clustering of buckets.
- As explained in Section 10.3, DDYF appears uniquely suited to fingerprint implementations: it surfaces benign behavior differences that are hard to find by other means and hard to detect with network-monitoring tools. We thus plan to build a complete fingerprinting methodology on top of DDYF.
- A further direction is to strengthen the differential oracle so that it additionally checks whether equality relations between randomized fields hold consistently on both sides, enabling detection of agreement violations on randomized data (*e.g.,* session keys) that occur on only one side.
- Fuzzing a new protocol with DDYF requires providing a blacklist of randomized fields to ignore; this step could be fully automated by running a differential campaign between two identical instances of the PUT.
- Finally, DDYF needs to be evaluated further, notably on other protocols and implementations (see the list in Section 11.3.6).

11.3.6. Engineering Infrastructure and Longer-Term Maintenance

Many of the research directions discussed above rely on substantial engineering investment in PUFFIN and its surrounding tooling. We plan to continue this important engineering effort —supported by research engineers O. Demengeon and N. Baffou— as we consider PUFFIN a primary research vector.

First, we plan to consolidate a robust infrastructure for the scientific evaluation of DYF and DDYF, and for the continuous evaluation of PUFFIN itself. Our central effort in this direction is PUFFIN-BENCH, a testing infrastructure —now largely complete[10]— that automates the reporting of performance, capabilities, and their regressions across commits and across new implementation ideas: it runs targeted vulnerability-finding campaigns and longer performance campaigns on selected PUTs, computes coverage and other key metrics, and produces statistical comparisons (means, confidence intervals, ablation curves) against a baseline commit. We plan to keep improving PUFFIN-BENCH, which we envision as a key piece of infrastructure for the long-term maintenance of PUFFIN and for the rigorous evaluation of future research directions.

10: https://github.com/tlspuffin/puffin-bench

Several further engineering tracks underpin our broader agenda: a

unified CMAKE build system for all harnessed implementations; full black-box fuzzing (*e.g.,* via QEMU-based coverage instrumentation) to extend our reach to closed-source PUTs; *diff-coverage* for fine-grained measurement of feedback improvements between versions; *etc.*

We also plan to apply DYF and DDYF to a wider range of targets. Additional TLS PUTs of interest include Microsoft's closed-source SSP SCHANNEL, the open-source MBEDTLS, Amazon's S2N-TLS, and BOTAN. Beyond covering TLS 1.2 and 1.3, it would also be valuable to target the closely related QUIC and DTLS protocols. TLS is arguably the most heavily tested cryptographic protocol, which made it both a challenging and a natural starting point for our DY fuzzing research; we are now eager to apply the approach to other protocols and implementations. In particular, we would like to test SIGNAL, WPA2 (and its successor WPA3), BLUETOOTH-related protocols, as well as closed-source targets such as those found in mobile telecommunication systems, IoT, and Industrial Control Systems (ICS). We also plan to enhance SSHPUFFIN for SSH and finalize OPCUAPUFFIN for OPC UA (the latter being the topic of V. Diemunsch's PhD).

11.4. Bridging the Gap Between DY Fuzzing and Verification

One of our long-term objectives is to develop a hybrid, synergistic framework combining two contrasting techniques: *implementation-aware DY verification* and *DY-model-guided fuzzing*. To carry out this, we have identified two key objectives that we detail below.

11.4.1. Hybrid Framework and Fuzzing

We will develop a hybrid framework where DY formal models and implementations are coupled and can drive each other via *dual executions*, where an abstract, symbolic state (in the protocol model) is assigned a concrete state (in the program) and vice versa.[11] The model expresses the *intended protocol design logic* that the PUT is intended to implement. We will leverage this hybrid framework to enhance our DY fuzzing approach and to develop *implementation-aware verification techniques* (see next subsection). This approach requires developing a new framework: from proposing formal foundations to implementing and evaluating on real-world codebases, and exploring the design space.

Our starting point and baseline for exploring this research direction is our recent DY fuzzer PUFFIN. We plan to refine policies, coverage, and *fit metrics* (as part of the feedback), notably by leveraging existing symbolic verifiers. Regarding policies, we will encode functional correctness with respect to the model and symbolic security goals as policies that can accommodate various properties and threat models to provide fine-grained analysis results. We will also define scoring metrics that can be effectively computed by symbolic verifiers and help the fuzzer promote test cases that are *close to attack traces*. Moreover, we will explore using verifiers to help synthesize test cases that increase coverage, in the spirit of concolic testing. We mentioned some of these objectives in Section 11.3 but this hybrid framework will allow us to achieve them in a more principled way and to explore more directions. Finally, formal guarantees on the implementation could be automatically obtained for bounded scenarios (number of sessions and message size), in the spirit of bounded model-checking [CD09].

11: Not to be confused with (dynamic) symbolic execution, whitebox fuzzing, hybrid fuzzing, or concolic testing (see *e.g.,* [SMA05]), which may seem similar to our hybrid testing on the surface but are radically different: the "symbolic" domains have nothing in common: a *protocol and DY adversary model* [Bar+21a] in our case versus an *arithmetic values model* for the above works.

11.4.2. Model Extraction and Refinement

We will develop a *lightweight annotation language* with formal semantics that practitioners can use in their programs, or that analysts can add to existing code, to express the intended protocol logic and from which protocol formal models are automatically extracted. The main challenge lies in designing the annotation language and the model extractor such that, for usability, only the core protocol logic needs to be user-provided while the rest should be directly extracted from the code through static and dynamic code analysis, in the spirit of [MK24; NKN23]. The goal is to compare the intent (symbolic model) against its implementation (code) from a symbolic model abstraction and security perspective.

It is likely that the PUT does not exactly comply with the declared model, and we expect our DY fuzzer to find concrete witnesses of functional correctness violations in such cases. We will develop techniques to compute model updates —*refinements*— from such concrete counterexamples, in the spirit of counterexample-guided abstraction refinement (CEGAR) [Cla+03; Cla+00]. The security impact of such discrepancies and model updates is then automatically assessed by symbolic verifiers. This feedback loop of counterexamples and model updates will narrow the gap between models and implementations and also capture potentially harmful implementation behaviors that were not initially intended.

We will validate and generalize the overall approach through its ability to find attacks and flaws in real-world protocol implementations such as the various internet protocols that are the backbone of many security systems (*e.g.,* WPA2, OpenSSH, OpenID, FIDO's U2F), closed-source yet critical protocols (*e.g.,* those running in baseband chips in phones or smart cards), or novel protocols motivated by emerging applications (*e.g.,* e-voting; see Section 11.2). We will start with our TLSPUFFIN test-bed on TLS stacks.

We close with a personal reflection on the conditions under which this research will be pursued.

The Toughest Challenge of All

The research directions outlined above will be pursued in a world that makes their pursuit increasingly difficult —not merely for technical reasons, but for deeper, structural ones.

The first challenge is to resist techno-centrism: to conduct research driven by scientific questions rather than by the momentum of technological trends, and to engage in genuine critical assessment of the systems we study, including their hidden risks and their potential for misuse. The values that brought us here must remain the compass.

The second is institutional. Scientific freedom is not unconditional; top-down policies can, and sometimes do, constrain it in ways that are at odds with open research. We intend to resist such pressures where we can, and to draw a clear line where we cannot.

The third is societal. In a world where scientific authority is increasingly contested, we see science communication and public engagement not as peripheral activities but as professional responsibilities. Making our results —and their limits— legible to non-specialists is part of the job, as we tried to do in our contact-tracing risk analysis [*Bon+20*] and in our

response to the CNIL e-voting recommendation [*Cor+25*].

Finally, and most acutely for our field: security research is increasingly instrumentalized by states and corporations for purposes of surveillance and control, often at the expense of the individual rights and civil liberties that motivate much of this work in the first place. We are committed to conducting this research in a way that is transparent about such tensions, that favors defensive and privacy-preserving applications, and that speaks out when the science is invoked to justify what the evidence does not support.

Bibliography

[Zub19] Shoshana Zuboff. *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*. PublicAffairs, 2019 (cited on page 2).

[Meh+14] Neel Mehta, Riku, Antti, and Matti. *The Heartbleed Bug*. CVE-2014-0160. https://heartbleed.com. 2014 (cited on pages 4, 5, 10).

[VP17] Mathy Vanhoef and Frank Piessens. "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2". In: *CCS*. ACM, 2017 (cited on page 4).

[BST21a] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *S&P*. IEEE, 2021 (cited on pages 4, 5, 52).

[CVE09-3555] *CVE-2009-3555*. https://www.cve.org/CVERecord?id=CVE-2009-3555 (cited on pages 4, 5, 149).

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *S&P*. IEEE, 2017 (cited on pages 4, 6–8).

[Cre+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *CCS*. ACM, 2017 (cited on pages 4, 7).

[Cre+16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *S&P*. IEEE Computer Society, 2016 (cited on page 4).

[Bha+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. "Implementing TLS with Verified Cryptographic Security". In: *S&P*. IEEE, 2013 (cited on pages 4, 157).

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE Bites: Exploiting the SSL 3.0 Fallback*. Tech. rep. Google, 2014 (cited on pages 4, 5, 149).

[Bha+14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *S&P*. IEEE, 2014 (cited on pages 4, 5).

[Avi+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, et al. "DROWN: Breaking TLS Using SSLv2". In: *USENIX Security Symposium*. USENIX Association, 2016 (cited on page 4).

[DG19] Nir Drucker and Shay Gueron. *Selfie: Reflections on TLS 1.3 with PSK*. ePrint IACR 347. 2019 (cited on page 4).

[Bor+19] Ravishankar Borgaonkar, Lucca Hirschi, Shinjo Park, and Altaf Shaik. "New Privacy Threat on 3G, 4G, and Upcoming 5G AKA Protocols". In: *Proc. Priv. Enhancing Technol.* 2019.3 (2019) (cited on pages 5, 11, 12, 52, 53, 99, 100, 109, 110, 112–115, 117, 119, 120).

[BST21b] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. "Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions". In: *USENIX Security Symposium*. 2021 (cited on pages 5, 52).

[BST23] David A. Basin, Patrick Schaller, and Jorge Toro-Pozo. "Inducing Authentication Failures to Bypass Credit Card PINs". In: *USENIX Security Symposium*. USENIX Association, 2023 (cited on page 5).

[CVE14-1266] *CVE-2014-1266*. https://www.cve.org/CVERecord?id=CVE-2014-1266 (cited on pages 5, 149, 156).

[Beu+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *S&P*. IEEE, 2015 (cited on pages 5, 149, 157, 166, 178).

[RP15] Joeri de Ruiter and Erik Poll. “Protocol State Fuzzing of TLS Implementations”. In: *USENIX Security Symposium*. USENIX Association, 2015 (cited on pages 5, 149, 157).

[Adr+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. “Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice”. In: *CCS*. ACM, 2015 (cited on page 5).

[Bei+21] Christof Beierle, Patrick Derbez, Gregor Leander, Gaëtan Leurent, Håvard Raddum, Yann Rotella, David Rupprecht, and Lukas Stennes. “Cryptanalysis of the GPRS Encryption Algorithms GEA-1 and GEA-2”. In: *EUROCRYPT (2)*. Lecture Notes in Computer Science. Springer, 2021 (cited on page 6).

[Tho84] Ken Thompson. “Reflections on Trusting Trust”. In: *Commun. ACM* 27.8 (1984) (cited on page 6).

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007 (cited on page 6).

[Can+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security Symposium*. USENIX Association, 2019 (cited on page 6).

[Bar+21a] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. “SoK: Computer-Aided Cryptography”. In: *S&P*. IEEE, 2021 (cited on pages 6, 35, 151, 157, 196).

[Bar+11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. “Computer-Aided Security Proofs for the Working Cryptographer”. In: *CRYPTO*. Lecture Notes in Computer Science. Springer, 2011 (cited on page 6).

[Bla08] Bruno Blanchet. “A Computationally Sound Mechanized Prover for Security Protocols”. In: *IEEE Trans. Dependable Secur. Comput.* 5.4 (2008) (cited on page 6).

[Bae+21] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. “An Interactive Prover for Protocol Verification in the Computational Model”. In: *S&P*. IEEE, 2021 (cited on page 6).

[Has+23] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *ACM Trans. Program. Lang. Syst.* 45.3 (2023) (cited on page 6).

[Swa+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. “Dependent types and multi-monadic effects in F*”. In: *POPL*. ACM, 2016 (cited on pages 6, 157).

[Cor+17] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. “Machine-Checked Proofs of Privacy for Electronic Voting Protocols”. In: *S&P*. IEEE, 2017 (cited on page 6).

[Del+21] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. “A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer”. In: *S&P*. IEEE, 2021 (cited on pages 7, 8, 157).

[DY83] Danny Dolev and Andrew Chi-Chih Yao. “On the security of public key protocols”. In: *IEEE Trans. Inf. Theory* 29.2 (1983) (cited on pages 7, 24, 157).

[Bla16] Bruno Blanchet. “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif”. In: *Found. Trends Priv. Secur.* 1.1-2 (2016) (cited on pages 7, 33, 158–160).

[Mei+13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *CAV*. Lecture Notes in Computer Science. Springer, 2013 (cited on pages 7, 105, 110, 118, 160).

[CKR18] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. “DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice”. In: *S&P*. IEEE, 2018 (cited on pages 7, 35, 55, 56).

[Che+22] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. "SAPIC+: protocol verifiers of the world, unite!" In: *USENIX Security Symposium*. USENIX Association, 2022 (cited on pages 7, 35).

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, eds. *Post-Quantum Cryptography*. Springer-Verlag Berlin Heidelberg, 2009 (cited on page 7).

[Che+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-Quantum Cryptography*. Tech. rep. NISTIR 8105. Gaithersburg, MD: National Institute of Standards and Technology, Apr. 2016 (cited on page 7).

[GCR23] Alexandre Augusto Giron, Ricardo Custódio, and Francisco Rodríguez-Henríquez. "Post-quantum hybrid key exchange: a systematic mapping study". In: *J. Cryptogr. Eng.* 13.1 (2023) (cited on page 7).

[Bha+24a] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. "Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging". In: *USENIX Security Symposium*. USENIX Association, 2024 (cited on pages 8, 187).

[Laf+25] Pascal Lafourcade, Dhekra Mahmoud, Sylvain Ruhault, and Abdul Rahman Taleb. "A Tale of Two Worlds, a Formal Story of WireGuard Hybridization". In: *USENIX Security Symposium*. USENIX Association, 2025 (cited on pages 8, 68, 187).

[Ame24] American National Institute of Standards and Technology (NIST), USA. *Transition to Post-Quantum Cryptography Standards*. NIST Internal Report (IR) 8547 (Initial Public Draft). 2024 (cited on page 8).

[Fre26] French National Agency for the Security of Information Systems (ANSSI). *Rules and Recommendations Concerning the Choice and Sizing of Cryptographic Mechanisms*. Tech. rep. 2026 (cited on page 8).

[Bar+21b] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. "EasyPQC: Verifying Post-Quantum Cryptography". In: *CCS*. ACM, 2021 (cited on page 8).

[CFJ22] Cas Cremers, Caroline Fontaine, and Charlie Jacomme. "A Logic and an Interactive Prover for the Computational Post-Quantum Security of Protocols". In: *S&P*. IEEE, 2022 (cited on page 8).

[Cel+22] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. "A Tale of Two Models: Formal Verification of KEMTLS via Tamarin". In: *ESORICS (3)*. Lecture Notes in Computer Science. Springer, 2022 (cited on page 8).

[LMR24] Pascal Lafourcade, Dhekra Mahmoud, and Sylvain Ruhault. "A Unified Symbolic Analysis of WireGuard". In: *NDSS*. The Internet Society, 2024 (cited on pages 8, 68, 187).

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. "Formal verification of security protocol implementations: a survey". In: *Formal Aspects Comput.* 26.1 (2014) (cited on page 8).

[Ahm+26] D. Ahman, Karthikeyan Bhargavan, Barry Bond, Jay Bosamiya, Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Aymeric Fromherz, Sydney Gibson, Chris Hawblitzel, et al. "Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software". In: *ACM Trans. Program. Lang. Syst.* 48.2 (2026) (cited on page 8).

[Zin+17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *CCS*. ACM, 2017 (cited on pages 8, 157).

[Bon+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code". In: *USENIX Security Symposium*. USENIX Association, 2017 (cited on page 8).

[Alm+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *S&P*. IEEE, 2020 (cited on pages 8, 157).

[CB13a] David Cadé and Bruno Blanchet. "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH". In: *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 4.1 (2013) (cited on page 8).

[CB15] David Cadé and Bruno Blanchet. "Proved Generation of Implementations from Computationally Secure Protocol Specifications". In: *J. Comput. Secur.* 23.3 (2015) (cited on page 8).

[SGP25] Pratap Singh, Joshua Gancher, and Bryan Parno. "OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries". In: *USENIX Security Symposium*. USENIX Association, 2025 (cited on page 8).

[Spr+20] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020) (cited on pages 8, 9).

[Arq+23] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. "Sound Verification of Security Protocols: From Design to Interoperable Implementations". In: *S&P*. IEEE, 2023 (cited on pages 8, 9, 157).

[Arq+26] Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller. "The Secrets Must Not Flow: Scaling Security Verification to Large Codebases". In: *S&P*. IEEE, 2026 (cited on pages 8, 9).

[Mod16] Paolo Modesti. "AnBx: Automatic Generation and Verification of Security Protocols Implementations". In: *Foundations and Practice of Security (FPS)*. Vol. 9482. Lecture Notes in Computer Science. Springer International Publishing, 2016 (cited on page 8).

[Sis+18] Riccardo Sisto, Piergiuseppe Bettassa Copet, Matteo Avalle, and Alfredo Pironti. "Formally Sound Implementations of Security Protocols with JavaSPI". In: *Formal Aspects Comput.* 30.2 (2018) (cited on page 8).

[AMV15] Omar Saad Almousa, Sebastian Mödersheim, and Luca Viganò. "Alice and Bob: Reconciling Formal Models and Implementation". In: *Programming Languages with Applications to Biology and Security*. Lecture Notes in Computer Science. Springer, 2015 (cited on page 8).

[GMF25] Rémi Garcia, Paolo Modesti, and Leo Freitas. "From Provable Models to Provable Implementations: Translating Alice & Bob Security Protocols to F*". In: *FPS*. 2025 (cited on page 8).

[BCF07] Karthikeyan Bhargavan, RJ Corin, and Cédric Fournet. *Crypto-verifying protocol implementations in ML*. Tech. rep. 2007 (cited on page 8).

[Bha+08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. "Verified Interoperable Implementations of Security Protocols". In: *ACM Trans. Program. Lang. Syst.* 31.1 (2008) (cited on page 8).

[BFK16] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. "miTLS: Verifying Protocol Implementations against Real-World Attacks". In: *IEEE Secur. Priv.* 14.6 (2016) (cited on page 8).

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *EuroS&P*. IEEE, 2017 (cited on page 8).

[AGJ11] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. "Extracting and verifying cryptographic models from C protocol code by symbolic execution". In: *CCS*. ACM, 2011 (cited on page 8).

[AGJ12] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. "Computational verification of C protocol implementations by symbolic execution". In: *CCS*. ACM, 2012 (cited on page 8).

[NKN23] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. "CryptoBap: A Binary Analysis Platform for Cryptographic Protocols". In: *CCS*. ACM, 2023 (cited on pages 8, 9, 197).

[VJ15] Gijs Vanspauwen and Bart Jacobs. "Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications". In: *SEFM*. Lecture Notes in Computer Science. Springer, 2015 (cited on page 8).

[Dup+14] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. "Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols". In: *J. Comput. Secur.* 22.5 (July 2014) (cited on page 8).

[Swa+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. "Secure distributed programming with value-dependent types". In: *ICFP*. ACM, 2011 (cited on page 8).

[Bha+21] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *EuroS&P*. IEEE, 2021 (cited on pages 8, 35, 157).

[Wal+23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. "TreeSync: Authenticated Group Management for Messaging Layer Security". In: *USENIX Security Symposium*. USENIX Association, 2023 (cited on pages 8, 9).

[Bha+24b] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. "hax: Verifying Security-Critical Rust Software Using Multiple Provers". In: *VSTTE*. Lecture Notes in Computer Science. Springer, 2024 (cited on page 8).

[Bha+25] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. "Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust". In: *CCS*. ACM, 2025 (cited on pages 8, 9).

[MK24] Kevin Morio and Robert Künnemann. "SpecMon: Modular Black-Box Runtime Monitoring of Security Protocols". In: *CCS*. ACM, 2024 (cited on pages 9, 192, 197).

[Man+21] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Trans. Software Eng.* 47.11 (2021) (cited on pages 9, 149).

[Zhu+22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. "Fuzzing: A Survey for Roadmap". In: *ACM Comput. Surv.* 54.11s (2022) (cited on page 9).

[Bab+19] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. "FUDGE: fuzz driver generation at scale". In: *ESEC/SIGSOFT FSE*. ACM, 2019 (cited on pages 9, 150).

[Vra20] Guido Vranken. *Cryptofuzz project*. https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/. 2020 (cited on pages 9, 156).

[Hir17] Lucca Hirschi. "Automated Verification of Privacy in Security Protocols: Back and Forth Between Theory & Practice (PhD thesis)". PhD thesis. Ecole Normale Supérieure de Paris-Saclay, 2017 (cited on pages 10, 55).

[PM16] Kenneth G. Paterson and Thyla van der Merwe. "Reactive and Proactive Standardisation of TLS". In: *SSR*. Lecture Notes in Computer Science. Springer, 2016 (cited on pages 10, 120).

[Che+23] Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. "Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses". In: *USENIX Security Symposium*. USENIX Association, 2023 (cited on pages 11, 40–42, 48, 49, 54, 58–60, 66, 68, 187).

[BHS19] David A. Basin, Lucca Hirschi, and Ralf Sasse. "Symbolic Analysis of Identity-Based Protocols". In: *Foundations of Security, Protocols, and Equational Reasoning*. Lecture Notes in Computer Science. Springer, 2019 (cited on pages 11, 49).

[HSB21] Lucca Hirschi, Lara Schmid, and David A. Basin. "Fixing the Achilles Heel of E-Voting: The Bulletin Board". In: *CSF*. IEEE, 2021 (cited on pages 11, 50, 51, 142).

[BDH17] David Baelde, Stéphanie Delaune, and Lucca Hirschi. "A Reduced Semantics for Deciding Trace Equivalence". In: *Log. Methods Comput. Sci.* 13.2 (2017) (cited on pages 11, 55).

[HC19] Lucca Hirschi and Cas Cremers. "Improving Automated Symbolic Analysis of Ballot Secrecy for E-Voting Protocols: A Method Based on Sufficient Conditions". In: *EuroS&P*. IEEE, 2019 (cited on pages 11, 55, 56, 188, 189).

[Dre+18] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. "Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR". In: *CSF*. IEEE Computer Society, 2018 (cited on pages 11, 52, 53, 105).

[Dre+20] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. "Verification of stateful cryptographic protocols with exclusive OR". In: *J. Comput. Secur.* 28.1 (2020) (cited on pages 11, 52, 53).

[Bas+18] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. "A Formal Analysis of 5G Authentication". In: *CCS*. ACM, 2018 (cited on pages 11, 12, 16, 52, 53, 73, 99–101, 104–106, 115, 117, 119, 120).

[Gir+20] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. "A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols". In: *USENIX Security Symposium*. USENIX Association, 2020 (cited on pages 11, 16, 20, 28, 58, 59, 61–64, 66–68, 187).

[DHK25a] Vincent Diemunsch, Lucca Hirschi, and Steve Kremer. "A Comprehensive Formal Security Analysis of OPC UA". In: *USENIX Security Symposium*. USENIX Association, 2025 (cited on pages 11, 12, 16, 39, 58–60, 64, 66, 68, 69, 73, 75, 76, 85, 93, 94, 97, 187).

[DH23] Alexandre Debant and Lucca Hirschi. "Reversing, Breaking, and Fixing the French Legislative Election E-Voting Protocol". In: *USENIX Security Symposium*. USENIX Association, 2023 (cited on pages 12, 122, 126, 127, 137, 140, 143, 144).

[AHK24] Max Ammann, Lucca Hirschi, and Steve Kremer. "DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing". In: *S&P*. IEEE, 2024 (cited on pages 13, 16, 150, 152, 153, 157, 158, 171, 174, 175, 180, 181).

[GHK27] Tom Gouville, Lucca Hirschi, and Steve Kremer. "DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols". In: *S&P*. IEEE, 2027 (cited on pages 13, 16, 150, 152, 159, 167–171, 180, 182, 186, 195).

[puffin] puffin team. *TLSPUFFIN: a Dolev-Yao model-guided fuzzer for cryptographic protocol implementations:* https://github.com/tlspuffin/tlspuffin (cited on pages 14, 150, 151, 171, 172, 176).

[Cor+24] Véronique Cortier, Alexandre Debant, Anselme Goetschmann, and Lucca Hirschi. "Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility". In: *USENIX Security Symposium*. USENIX Association, 2024 (cited on pages 16, 140, 141).

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *IEEE Trans. Inf. Theory* 22.6 (1976) (cited on page 18).

[RSA78] R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (1978) (cited on page 18).

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH". In: *NDSS*. The Internet Society, 2016 (cited on pages 20, 41–43, 48).

[Kra03] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols". In: *CRYPTO*. Lecture Notes in Computer Science. Springer, 2003 (cited on page 20).

[Per18] Trevor Perrin. *The Noise Protocol Framework*. Revision 34, https://noiseprotocol.org/noise.html. 2018 (cited on pages 20, 23, 24, 67).

[Gir19] Guillaume Girol. "Formalizing and Verifying the Security Protocols from the Noise Framework". https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/noise-ggirol.zip. MA thesis. ETH Zurich, 2019 (cited on pages 22, 61, 63–66).

[Don17] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *NDSS*. The Internet Society, 2017 (cited on page 23).

[Inc17] WhatsApp Inc. *WhatsApp encryption overwiew—Technical white paper*. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. 2017 (cited on page 23).

[Net17] The Lightning Network. *BOLT 8: Encrypted and Authenticated Transport*. https://github.com/lightningnetwork/lightning-rfc/blob/130bc5da2c05f212fba09ae309e53fec8cde2c6d/08-transport.md. 2017 (cited on page 23).

[Low97] Gavin Lowe. "A Hierarchy of Authentication Specification". In: *CSFW*. IEEE Computer Society, 1997 (cited on pages 24, 82).

[AF01] Martín Abadi and Cédric Fournet. "Mobile values, new names, and secure communication". In: *POPL*. ACM, 2001 (cited on pages 24–26, 33, 188).

[Sch+12a] Benedikt Schmidt, Simon Meier, Cas Cremers, and David A. Basin. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *CSF*. IEEE Computer Society, 2012 (cited on pages 24, 26, 27, 29, 31–33).

[Sch12] Benedikt Schmidt. “Formal analysis of key exchange protocols and physical protocols”. PhD thesis. ETH Zurich, Zürich, Switzerland, 2012 (cited on pages 24, 26, 29, 30, 32, 33).

[Mei13] Simon Meier. “Advancing automated security protocol verification”. PhD thesis. ETH Zurich, Zürich, Switzerland, 2013 (cited on pages 24, 26, 29, 33, 53, 54).

[CD05] Hubert Comon-Lundh and Stéphanie Delaune. “The Finite Variant Property: How to Get Rid of Some Algebraic Properties”. In: *RTA*. Lecture Notes in Computer Science. Springer, 2005 (cited on pages 25, 30, 31).

[BDS15] David A. Basin, Jannik Dreier, and Ralf Sasse. “Automated Symbolic Proofs of Observational Equivalence”. In: *CCS*. ACM, 2015 (cited on pages 28, 53).

[Hüt03] Hans Hüttel. “Deciding framed bisimilarity”. In: *Electron. Notes Theor. Comput. Sci.* 68.6 (2003) (cited on page 28).

[CCD15] Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. “From security protocols to pushdown automata”. In: *ACM Trans. Comput. Log.* 17.1:3 (Sept. 2015) (cited on page 28).

[ESM12] Santiago Escobar, Ralf Sasse, and José Meseguer. “Folding variant narrowing and optimal variant termination”. In: *J. Log. Algebraic Methods Program.* 81.7-8 (2012) (cited on pages 31, 53).

[Cla+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007 (cited on page 31).

[Sch+12b] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. *Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties (Extended Version 1, April 24th, 2012)*. https://infsec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/dh_tamarin_extended_v1.pdf. 2012 (cited on page 31).

[DKK26] Jannik Dreier, Elise Klein, and Steve Kremer. “Tamarin Unchained: Handling User-Defined AC Operators”. In: *CSF*. IEEE, 2026 (cited on pages 32, 53).

[Dre+17] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. “Beyond Subterm-Convergent Equational Theories in Automated Verification of Stateful Protocols”. In: *POST*. Lecture Notes in Computer Science. Springer, 2017 (cited on page 32).

[EMM09] Santiago Escobar, Catherine Meadows, and José Meseguer. “Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties”. In: *Foundations of Security Analysis and Design V*. Vol. 5705. LNCS. Springer, 2009 (cited on page 35).

[San+14] Sonia Santiago, Santiago Escobar, Catherine Meadows, and José Meseguer. “A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA”. In: *STM*. Lecture Notes in Computer Science. Springer, 2014 (cited on page 35).

[TD10] Alwen Tiu and Jeremy E. Dawson. “Automating Open Bisimulation Checking for the Spi Calculus”. In: *CSF*. IEEE Computer Society, 2010 (cited on pages 35, 55).

[Cha+16] Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. “Automated Verification of Equivalence Properties of Cryptographic Protocols”. In: *ACM Trans. Comput. Log.* 17.4 (2016) (cited on pages 35, 55).

[GK17] Ivan Gazeau and Steve Kremer. “Automated analysis of equivalence properties for security protocols using else branches”. In: *ESORICS*. Springer. 2017 (cited on page 35).

[Che14] Vincent Cheval. “APTE: An Algorithm for Proving Trace Equivalence”. In: *TACAS*. Lecture Notes in Computer Science. Springer, 2014 (cited on pages 35, 56).

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996 (cited on page 41).

[BR93] Mihir Bellare and Phillip Rogaway. “Random Oracles are Practical: A Paradigm for Designing Efficient Protocols”. In: *CCS*. ACM, 1993 (cited on page 41).

[Tsu92] Gene Tsudik. “Message authentication with one-way hash functions”. In: *Comput. Commun. Rev.* 22.5 (1992) (cited on page 41).

[Aur17] Valerie Aurora. *Lifetimes of cryptographic hash functions*. https://valerieaurora.org/hash.html. 2017 (cited on pages 41–43).

[Wil17] Zooko Wilcox. *Lessons From The History Of Attacks On Secure Hash Functions*. https://electriccoin.co/blog/lessons-from-the-history-of-attacks-on-secure-hash-functions/. 2017 (cited on page 41).

[Ste21] Marc Stevens. “A Survey of Chosen-Prefix Collision Attacks”. In: *Computational Cryptography: Algorithmic Aspects of Cryptology*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2021 (cited on pages 41–44).

[SLW07] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. “Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities”. In: *EUROCRYPT*. Lecture Notes in Computer Science. Springer, 2007 (cited on pages 42–44).

[Duo09] Thai Duong. *Flickr’s API signature forgery vulnerability*. https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html. 2009 (cited on page 42).

[LP20] Gaëtan Leurent and Thomas Peyrin. “SHA-1 is a Shambles: First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust”. In: *USENIX Security Symposium*. USENIX Association, 2020 (cited on pages 43, 44).

[SA09] Yu Sasaki and Kazumaro Aoki. “Finding Preimages in Full MD5 Faster Than Exhaustive Search”. In: *EUROCRYPT*. Lecture Notes in Computer Science. Springer, 2009 (cited on page 43).

[Sas+07] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. “New Message Difference for MD4”. In: *FSE*. Lecture Notes in Computer Science. Springer, 2007 (cited on page 43).

[XLF13] Tao Xie, Fanbao Liu, and Dengguo Feng. “Fast Collision Attack on MD5”. In: *IACR Cryptol. ePrint Arch.* 2013 (2013) (cited on page 43).

[ZL12] Jinmin Zhong and Xuejia Lai. “Improved preimage attack on one-block MD4”. In: *J. Syst. Softw.* 85.4 (2012) (cited on page 43).

[Hir19] Lucca Hirschi. “Symbolic Abstractions for Quantum Protocol Verification”. In: *CoRR* abs/1904.04186 (2019) (cited on page 49).

[Sha84] Adi Shamir. “Identity-Based Cryptosystems and Signature Schemes”. In: *CRYPTO*. Lecture Notes in Computer Science. Springer, 1984 (cited on page 49).

[CM06] Dominique Cansell and Dominique Mery. *Tutorial on the event-based B method*. https://cel.archives-ouvertes.fr/inria-00092846. 2006 (cited on page 50).

[Cou+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. “TLA + Proofs”. In: *FM*. Lecture Notes in Computer Science. Springer, 2012 (cited on page 50).

[Hop+16] Daira Hopwood, Sean Bowe, Taylor Hornby, Nathan Wilcox, et al. *Zcash protocol specification*. Technical report. 2016 (cited on page 51).

[Ben+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *S&P*. IEEE Computer Society, 2014 (cited on page 51).

[Bas+24] David A. Basin, Xenia Hofmeier, Ralf Sasse, and Jorge Toro-Pozo. “Getting Chip Card Payments Right”. In: *FM (1)*. Lecture Notes in Computer Science. Springer, 2024 (cited on page 52).

[WZX21] Yuchen Wang, Zhenfeng Zhang, and Yongquan Xie. “Privacy-Preserving and Standard-Compatible AKA Protocol for 5G”. In: *USENIX Security Symposium*. USENIX Association, 2021 (cited on pages 52, 117, 120).

[NSB21] Karl Norrman, Vaishnavi Sundararajan, and Alessandro Bruni. “Formal Analysis of EDHOC Key Establishment for Constrained IoT Devices”. In: *SECRYPT*. SCITEPRESS, 2021 (cited on page 52).

[DDW22] Alexandre Debant, Stéphanie Delaune, and Cyrille Wiedling. “So Near and Yet So Far - Symbolic Verification of Distance-Bounding Protocols”. In: *ACM Trans. Priv. Secur.* 25.2 (2022) (cited on page 52).

[CD19] Cas Cremers and Martin Dehnel-Wild. “Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion”. In: *NDSS*. The Internet Society, 2019 (cited on pages 52, 107, 117).

[Dur+18] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. “Associative Unification and Symbolic Reasoning Modulo Associativity in Maude”. In: *WRLA*. Lecture Notes in Computer Science. Springer, 2018 (cited on page 54).

[BDH18] David Baelde, Stéphanie Delaune, and Lucca Hirschi. “POR for Security Protocol Equivalences - Beyond Action-Determinism”. In: *ESORICS (1)*. Lecture Notes in Computer Science. Springer, 2018 (cited on page 55).

[AF04] Martín Abadi and Cédric Fournet. “Private authentication”. In: *Theor. Comput. Sci.* 322.3 (2004) (cited on page 55).

[Pel98] Doron A. Peled. “Ten Years of Partial Order Reduction”. In: *CAV*. Lecture Notes in Computer Science. Springer, 1998 (cited on page 55).

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. Lecture Notes in Computer Science. Springer, 1996 (cited on pages 55, 56).

[CJM03] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. “Efficient verification of security protocols using partial-order reductions”. In: *Int. J. Softw. Tools Technol. Transf.* 4.2 (2003) (cited on page 55).

[BDH14] David Baelde, Stéphanie Delaune, and Lucca Hirschi. “A Reduced Semantics for Deciding Trace Equivalence Using Constraint Systems”. In: *POST*. Lecture Notes in Computer Science. Springer, 2014 (cited on page 55).

[BDH15] David Baelde, Stéphanie Delaune, and Lucca Hirschi. “Partial Order Reduction for Security Protocols”. In: *CONCUR*. 2015 (cited on page 55).

[CB13b] Vincent Cheval and Bruno Blanchet. “Proving More Observational Equivalences with ProVerif”. In: *Proc. 2nd Conference on Principles of Security and Trust*. Vol. 7796. LNCS. Springer, 2013 (cited on page 56).

[Ara+10] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. “Analysing Unlinkability and Anonymity Using the Applied Pi Calculus”. In: *CSF*. IEEE, 2010 (cited on page 56).

[BS16] Bruno Blanchet and Ben Smyth. “Automated Reasoning for Equivalences in the Applied Pi Calculus with Barriers”. In: *CSF*. IEEE Computer Society, 2016 (cited on page 56).

[HBD16] Lucca Hirschi, David Baelde, and Stéphanie Delaune. “A Method for Verifying Privacy-Type Properties: The Unbounded Case”. In: *S&P*. IEEE, 2016 (cited on pages 56, 188, 189).

[HBD19] Lucca Hirschi, David Baelde, and Stéphanie Delaune. “A method for unbounded verification of privacy-type properties”. In: *J. Comput. Secur.* 27.3 (2019) (cited on pages 56, 188, 189).

[BDM20] David Baelde, Stéphanie Delaune, and Solène Moreau. “A Method for Proving Unlinkability of Stateful Protocols”. In: *CSF*. IEEE, 2020 (cited on page 56).

[BC14] David A. Basin and Cas Cremers. “Know Your Enemy: Compromising Adversaries in Protocol Analysis”. In: *ACM Trans. Inf. Syst. Secur.* 17.2 (2014) (cited on pages 58, 59, 68).

[Mah25] Dhekra Mahmoud. “Security Protocol Design and Symbolic Analysis: Hybrid Protocols, Derived Adversary Models, and Refined Equational Theories”. PhD thesis. Université Clermont Auvergne, 2025 (cited on pages 59, 68).

[Jac+23] Charlie Jacomme, Elise Klein, Steve Kremer, and Maïwenn Racouchot. “A comprehensive, formal and automated analysis of the EDHOC protocol”. In: *USENIX Security Symposium*. USENIX Association, 2023 (cited on page 68).

[Bas+19] David Basin, Cas Cremers, Jannik Dreier, Sasa Radomirovic, Ralf Sasse, Lara Schmid, and Benedikt Schmidt. *The Tamarin Manual*. https://tamarin-prover.github.io/manual/. 2019 (cited on page 68).

[BSC16] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2016 (cited on page 68).

[DHK25b] Vincent Diemunsch, Lucca Hirschi, and Steve Kremer. *Submission artifacts*. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/vdh-anssi/opc-ua_security. 2025 (cited on pages 70, 76, 83, 86).

[CGG19] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. "Belenios: A Simple Private and Verifiable Electronic Voting System". In: *Foundations of Security, Protocols, and Equational Reasoning*. Lecture Notes in Computer Science. Springer, 2019 (cited on pages 73, 122, 124–126, 139, 140, 142).

[BC12] David A. Basin and Srdjan Capkun. "The research value of publishing attacks". In: *Commun. ACM* 55.11 (2012) (cited on page 74).

[Sto+23] Keith Stouffer, Michael Pease, CheeYee Tang, Timothy Zimmerman, Victoria Pillitteri, Suzanne Lightman, Adam Hahn, Stephanie Saravia, Aslam Sherule, and Michael Thompson. *Guide to Operational Technology (OT) Security*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r3.pdf. 2023 (cited on pages 75, 78).

[BR17] Marie Baezner and Patrice Robin. *Stuxnet*. http://hdl.handle.net/20.500.11850/200661. 2017 (cited on page 75).

[OSS16] Sebastian Obermeier, Roman Schlegel, and Johannes Schneider. "A security evaluation of IEC 62351". In: *Journal of Information Security and Applications*. 2016 (cited on page 75).

[Dam+17] Damm, Gappmeier, Zugfil, Plöb, Fiat, and Störtkuhl. *OPC-UA security analysis*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik (BSI), 2017 (cited on page 75).

[Dor+22] Johannes vom Dorp, Sven Merschjohann, David Meier, Florian Patzer, Markus Karch, and Christian Haas. *OPC-UA security analysis*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik (BSI), 2022 (cited on page 75).

[Sys23] Systerel. *A safe and secure OPC UA implementation*. https://www.s2opc.com/. Version 1.3.1. 2023 (cited on page 75).

[CT18] Pavel Cheremushkin and Sergey Temnikov. *OPC UA security analysis*. Ed. by Kaspersky Lab ICS CERT. https://ics-cert.kaspersky.com/publications/reports/2018/05/10/opc-ua-security-analysis/. 2018 (cited on page 75).

[Tea23] Team82 Research. *OPC UA Deep Dive: A Complete Guide to the OPC UA Attack Surface*. Ed. by Claroty. https://claroty.com/team82/research/opc-ua-deep-dive-a-complete-guide-to-the-opc-ua-attack-surface. 2023 (cited on page 75).

[BM23] Sharon Brizinov and Noam Moshe. *Exploiting OPC-UA in Every Possible Way: Practical Attacks Against Modern OPC-UA Architectures*. https://i.blackhat.com/BH-US-23/Presentations/US-23-BRIZINOV-Exploiting-OPC.pdf. 2023 (cited on page 75).

[Bla01] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *CSFW*. IEEE, 2001 (cited on pages 76, 118).

[BCC22] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More". In: *S&P*. IEEE, 2022 (cited on page 76).

[OPC23] OPC Foundation. *OPC Unified Architecture Specification*. https://reference.opcfoundation.org/. Release 1.05.03. 2023 (cited on pages 77, 81, 82, 85, 86, 93).

[DHK25c] Vincent Diemunsch, Lucca Hirschi, and Steve Kremer. *A Comprehensive Formal Security Analysis of OPC UA, full report*. https://eprint.iacr.org/2025/148. 2025 (cited on pages 78, 79, 81, 82, 87, 89, 90, 92–95, 97).

[OPC24] OPC Foundation. *OPC Unified Architecture Specification: Profiles*. https://profiles.opcfoundation.org/profile/. Release 1.05. 2024 (cited on page 78).

[BCW22] Karthikeyan Bhargavan, Vincent Cheval, and Christopher A. Wood. "A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello". In: *CCS*. ACM, 2022 (cited on page 83).

[Low96] Gavin Lowe. "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1996 (cited on page 86).

[Res18] Eric Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.3". In: *RFC* 8446 (2018) (cited on pages 88, 154).

[Ter25] Tom Tervoort. *No VPN Needed? Cryptographic Attacks Against the OPC UA Protocol*. https://i.blackhat.com/BH-USA-25/Presentations/USA-25-Tervoort-No-VPN-Needed-Cryptographic-Attacks-wp.pdf. 2025 (cited on page 93).

[OPC25] OPC Foundation. *OPC Unified Architecture Specification*. https://profiles.opcfoundation.org/documentversion/282. Release Candidate 1.05.07. 2025 (cited on page 94).

[DHK] Vincent Diemunsch, Lucca Hirschi, and Steve Kremer. *Vulnerability Report Disclosed to the OPC UA Foundation*. A copy can be found at https://github.com/vdh-anssi/opc-ua_security/blob/main/vulnerabilities.pdf (cited on pages 95–97).

[OBH17] Piers O'Hanlon, Ravishankar Borgaonkar, and Lucca Hirschi. "Mobile Subscriber WiFi Privacy". In: *S&P Workshops*. IEEE, 2017 (cited on pages 99, 100, 109, 117, 119).

[GSM17] GSMA. *Global Mobile Trends 2017*. https://www.gsma.com/globalmobiletrends/. 2017 (cited on page 99).

[GSM26] GSMA. *The Mobile Economy 2026*. https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/. 2026 (cited on page 99).

[3GP18] 3GPP. *Security architecture and procedures for 5G system*. TS 33.501, v15.1.0. 2018 (cited on pages 99, 100).

[3GPa] 3GPP. *Service requirements for the Evolved Packet System (EPS)*. TS 122.278. (3GPP) (cited on page 99).

[3GPb] 3GPP. *3G security; Security architecture*. TS 33.102. (3GPP) (cited on pages 99, 110).

[Ara+12] Myrto Arapinis, Loretta Ilaria Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. "New privacy issues in mobile telephony: fix and verification". In: *CCS*. ACM, 2012 (cited on pages 99, 100).

[Hah+14] Changhee Hahn, Hyunsoo Kwon, Daeyoung Kim, Kyungtae Kang, and Junbeom Hur. "A Privacy Threat in 4th Generation Mobile Telephony and Its Countermeasure". In: *WASA*. Lecture Notes in Computer Science. Springer, 2014 (cited on pages 99, 100).

[Sha+16] Altaf Shaik, Jean-Pierre Seifert, Ravishankar Borgaonkar, N. Asokan, and Valtteri Niemi. "Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems". In: *NDSS*. The Internet Society, 2016 (cited on pages 99, 100, 114, 115).

[UB] Swapnil Udar and Ravishankar Borgaonkar. *Understanding IMSI Privacy*. https://www.isti.tu-berlin.de/fileadmin/fg214/ravi/Darshak-bh14.pdf (cited on pages 99, 100).

[BVR15] Fabian van den Broek, Roel Verdult, and Joeri de Ruiter. "Defeating IMSI Catchers". In: *CCS*. ACM, 2015 (cited on pages 99, 100).

[KM14] Mohammed Shafiul Alam Khan and Chris J. Mitchell. "Another Look at Privacy Threats in 3G Mobile Telephony". In: *ACISP*. Lecture Notes in Computer Science. Springer, 2014 (cited on pages 99, 100).

[ZF05] Muxiang Zhang and Yuguang Fang. "Security analysis and enhancements of 3GPP authentication and key agreement protocol". In: *IEEE Trans. Wirel. Commun.* 4.2 (2005) (cited on pages 99, 100).

[KAC08] Muzammil Khan, Attiq Ahmed, and Ahmad Raza Cheema. "Vulnerabilities of UMTS Access Domain Security Architecture". In: *SNPD*. IEEE Computer Society, 2008 (cited on pages 99, 100).

[BS13] Anastasios N. Bikos and Nicolas Sklavos. "LTE/SAE Security Issues on 4G Wireless Networks". In: *IEEE Secur. Priv.* 11.2 (2013) (cited on pages 99, 100).

[3GPc] 3GPP. *Security architecture and procedures for 5G System*. TS 33.501. (3GPP) (cited on pages 100, 110).

[AH06] Jari Arkko and Henry Haverinen. "Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA)". In: *RFC* 4187 (2006) (cited on page 100).

[Pet] Peter Howard. *AKA usage in 3GPP*. URL: http://www.3gpp.org/ftp/tsg_sa/wg3_security/tsgs3_34_acapulco/docs/PDF/S3-040645.pdf (cited on page 100).

[ME12] A. B. M. Musa and Jakob Eriksson. "Tracking unmodified smartphones using wi-fi monitors". In: *SenSys*. ACM, 2012 (cited on page 113).

[CH13] Stephanie Clifford and Quentin Hardy. *Attention Shoppers: Store Is Tracking Your Cell*. New York Times. 2013 (cited on page 113).

[Ett] Ettus Research. *USRP B210*. URL: https://www.ettus.com/product/details/UB210-KIT (cited on page 113).

[PCS] PC/SC Workgroup. *PC/SC Workgroup Specifications*. URL: http://www.pcscworkgroup.com/specifications/specdownload.php (cited on page 113).

[Adv] Advanced Card Systems Holdings Limited. *ACR38 Smart Card Reader*. URL: http://www.acs.com.hk/en/products/4/acr38-smart-card-reader/ (cited on page 113).

[Gom+16] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Douglas J. Leith. "srsLTE: an open-source platform for LTE evolution and experimentation". In: *WiNTECH*. ACM, 2016 (cited on page 113).

[Sys] Sysmocom. *sysmoUSIM-SJS1*. URL: https://www.sysmocom.de/products/sysmousim-sjs1-sim-usim/index.html (cited on page 114).

[Woj] Ben Wojtowicz. *OpenLTE*. https://sourceforge.net/projects/openlte/ (cited on page 114).

[Eng] Tobias Engel. *Locating Mobile Phones using Signalling System 7*. https://berlin.ccc.de/~tobias/25c3-locating-mobile-phones.pdf (cited on page 114).

[Hus+19] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. "Privacy Attacks to the 4G and 5G Cellular Paging Protocols Using Side Channel Information". In: *NDSS*. The Internet Society, 2019 (cited on page 117).

[EAL20] Ed Kamya Kiyemba Edris, Mahdi Aiash, and Jonathan Kok-Keng Loo. "Formal Verification and Analysis of Primary Authentication based on 5G-AKA Protocol". In: *SDS*. IEEE, 2020 (cited on page 117).

[FOR16] Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. "Achieving Better Privacy for the 3GPP AKA Protocol". In: *Proc. Priv. Enhancing Technol.* 2016.4 (2016) (cited on page 117).

[Kou19] Adrien Koutsos. "The 5G-AKA Authentication Protocol Privacy". In: *EuroS&P*. IEEE, 2019 (cited on page 117).

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *FOCS*. IEEE Computer Society, 2001 (cited on page 118).

[MV19] Sebastian Mödersheim and Luca Viganò. "Alpha-Beta Privacy". In: *ACM Trans. Priv. Secur.* 22.1 (2019) (cited on page 118).

[BH17] R. Borgaonkar and L. Hirshi. *New Attacks in 4G Technologies*. Invited talk at Troopers. 2017. URL: https://www.troopers.de/troopers17/telco-sec-day/ (cited on page 119).

[Bor+17] Ravishankar Borgaonkar, Lucca Hirshi, Shinjo Park, Altaf Shaik, Andrew Martin, and Jean-Pierre Seifert. *New Adventures in Spying 3G & 4G Users: Locate, Track, Monitor*. Briefing at BlackHat USA 2017. 2017. URL: https://www.blackhat.com/us-17/briefings.html#new-adventures-in-spying-3g-and-4g-users-locate-track-and-monitor (cited on page 119).

[Gam] Gamry Instrumens. *Faraday Cage: What Is It? How Does It Work?* URL: http://www.gamry.com/application-notes/instrumentation/faraday-cage/ (cited on page 120).

[HSD19] Lucca Hirschi, Ralf Sasse, and Jannik Dreier. "Security Issues in the 5G Standard and How Formal Methods Come to the Rescue". In: *ERCIM News* 2019.117 (2019) (cited on page 120).

[Bas+15] David A. Basin, Cas Cremers, Kunihiko Miyazaki, Sasa Radomirovic, and Dai Watanabe. "Improving the Security of Cryptographic Protocol Standards". In: *IEEE Secur. Priv.* 13.3 (2015) (cited on page 120).

[Li+25] Hui Li, Haotian Li, Chi Ma, Jingjing Guan, Junchi Zeng, Haonan Feng, and Ziming Zhao. "5G-RNAKA : A Random Number-based Authentication and Key Agreement Protocol for 5G Systems". In: *CCS*. ACM, 2025 (cited on page 120).

[JCJ05] Ari Juels, Dario Catalano, and Markus Jakobsson. "Coercion-resistant electronic elections". In: *WPES*. ACM, 2005 (cited on pages 122, 139).

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. "Civitas: Toward a Secure Voting System". In: *S&P*. IEEE, 2008 (cited on pages 122, 139, 142).

[Adi08] Ben Adida. "Helios: Web-based Open-Audit Voting". In: *USENIX Security Symposium*. USENIX Association, 2008 (cited on pages 122, 125, 126, 142).

[flep22-r1] Results of the first round of the French Legislative elections 2022: https://www.diplomatie.gouv.fr/fr/services- aux-francais/voter-a-l-etranger/resultats- des-elections/article/elections- legislatives-resultats-du-1er-tour-pour- les-francais-de-l-etranger (cited on page 122).

[Doc23] Voxaly Docaposte. *Partial specification of the FLEP*. Available at the link https://www.voxaly.com/wp-content/uploads/VOXALY-LEG2023-Transparence-et-Verifiabilite-Specifications-publiques-v2-04.pdf obtained from https://www.voxaly.com/vote-par-internet-pour-les-francais-de-letranger-dans-le-cadre-des-elections-legislatives-partielles-2023. 2023 (cited on pages 124, 127, 134, 143).

[Mül22] Johannes Müller. "Breaking and Fixing Vote Privacy of the Estonian E-Voting Protocol IVXV". In: *Financial Cryptography Workshops*. Lecture Notes in Computer Science. Springer, 2022 (cited on page 124).

[Hai+20] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. "How not to prove your election outcome". In: *S&P*. IEEE, 2020 (cited on page 124).

[GG20] Pierrick Gaudry and Alexander Golovnev. "Breaking the Encryption Scheme of the Moscow Internet Voting System". In: *Financial Cryptography*. Lecture Notes in Computer Science. Springer, 2020 (cited on page 124).

[Doc22] Voxaly Docaposte. *Partial specification of the FLEP*. Available at the link https://w8t9w2j6.stackpathcdn.com/wp-content/uploads/VOXALY_LEG2022_Verifiabilite_Specifications.pdf obtained from https://www.voxaly.com/vote-par-internet-pour-les-francais-de-letranger-dans-le-cadre-des-elections-legislatives-2022. 2022 (cited on pages 124–127, 129–131, 142).

[CGGa] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. *Informal description of the Vvfe web services*. In 2022: https://verifiabilite-legislatives2022.fr/. In 2023: https://verifiabilite-legislatives2023.fr/ (cited on pages 125, 127, 130).

[CGGb] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. *Vvfe: Vérifiabilité du Vote des Français de l'Étranger*. https://gitlab.inria.fr/vvfe/vvfe (cited on page 125).

[éle] Code électoral. *French law governing political elections*. https://www.legifrance.gouv.fr/codes/section_lc/LEGITEXT000006070239/LEGISCTA000006115482 (cited on pages 125, 126).

[Off19] Journal Officiel. *Délibération n° 2019-053 du 25 avril 2019*. Available at the link https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000038661239. 2019 (cited on pages 125, 126).

[DH22] Alexandre Debant and Lucca Hirschi. *Reversing, Breaking, and Fixing the French Legislative Election E-Voting Protocol*. Cryptology ePrint Archive, Paper 2022/1653. https://eprint.iacr.org/2022/1653. 2022. URL: https://eprint.iacr.org/2022/1653 (cited on pages 125, 128, 130, 131, 133, 136, 137, 139–141, 143).

[Hei+16] Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. "Improving the Verifiability of the Estonian Internet Voting Scheme". In: *E-VOTE-ID*. Lecture Notes in Computer Science. Springer, 2016 (cited on page 126).

[Swi] Swiss Post. *e-voting system*. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation (cited on page 126).

[flep22-test] Large scale test of the FLEP in 2022: https://amsterdam.consulfrance.org/ Elections-legislatives-2022-vote-par- internet-second-test-grandeur-nature (cited on page 127).

[sessStorage] Firefox sessionStorage documentation: https://developer.mozilla.org/fr/docs/Web/API/Window/sessionStorage (cited on page 129).

[KTV12] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. "Clash Attacks on the Verifiability of E-Voting Systems". In: *S&P*. IEEE, 2012 (cited on page 132).

[MMR23] David Mestel, Johannes Müller, and Pascal Reisert. “How efficient are replay attacks against vote privacy? A formal quantitative analysis”. In: *J. Comput. Secur.* 31.5 (2023) (cited on page 137).

[flep22-r2] Results of the second round of the French Legislative elections 2022: https://www.diplomatie.gouv.fr/fr/services- aux-francais/voter-a-l-etranger/resultats- des-elections/article/elections- legislatives-resultats-du-2eme-tour-pour- les-francais-de-l-etranger (cited on page 137).

[CGGc] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. wip *branch (commit 5251b2f2) of the Vvfe tool*. https://gitlab.inria.fr/vvfe/vvfe (cited on page 138).

[Riv08] Ronald L Rivest. “On the notion of ”software independence” in voting systems”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366.1881 (2008) (cited on page 139).

[Bel+13] Susan Bell, Josh Benaloh, Michael D. Byrne, Dana DeBeauvoir, Bryce Eakin, Gail Fisher, Philip Kortum, Neal McBurnett, Julian Montoya, Michelle Parker, et al. “STAR-Vote: A Secure, Transparent, Auditable, and Reliable Voting System”. In: *EVT/WOTE*. USENIX Association, 2013 (cited on page 139).

[Ben87] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. Yale University, 1987 (cited on page 139).

[CL18] Véronique Cortier and Joseph Lallemand. “Voting: You Can’t Have Privacy without Individual Verifiability”. In: *CCS*. ACM, 2018 (cited on page 140).

[CDG21] Véronique Cortier, Alexandre Debant, and Pierrick Gaudry. *A privacy attack on the Swiss Post e-voting system*. Research Report. https://hal.inria.fr/hal-03446801/file/rwc.pdf. Université de Lorraine, CNRS, Inria, LORIA, Nov. 2021 (cited on page 140).

[Sak+14] Natsuhiko Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. “Openid connect core 1.0”. In: *The OpenID Foundation* (2014). https://openid.net/specs/openid-connect-core-1_0.html (cited on page 140).

[Chu+07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. “Attested append-only memory: making adversaries stick to their word”. In: *SOSP*. ACM, 2007 (cited on page 142).

[RKS15] Tim Ruffing, Aniket Kate, and Dominique Schröder. “Liar, Liar, Coins on Fire!: Penalizing Equivocation By Loss of Bitcoins”. In: *CCS*. ACM, 2015 (cited on page 142).

[Cor+25] Véronique Cortier, Alexandre Debant, Jannik Dreier, Pierrick Gaudry, Lucca Hirschi, and Steve Kremer. *Réponse au projet de mise à jour de la recommandation de la CNIL sur le vote électronique*. https://inria.hal.science/hal-04971713/. 2025 (cited on pages 144, 198).

[Fit+20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. “Analysis of DTLS Implementations Using Protocol State Fuzzing”. In: *USENIX Security Symposium*. USENIX Association, 2020 (cited on pages 149, 157, 174).

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. “Return Of Bleichenbacher’s Oracle Threat (ROBOT)”. In: *USENIX Security Symposium*. USENIX Association, 2018 (cited on page 149).

[RLD22] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. “Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks”. In: *ESORICS*. Springer, 2022 (cited on pages 149, 157, 176–178, 182).

[Fio+20] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *WOOT @ USENIX Security Symposium*. USENIX Association, 2020 (cited on pages 149, 156, 175, 177).

[Zal16] Michał Zalewski. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt. 2016 (cited on pages 149, 156, 177).

[LLV] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html (cited on pages 149, 156, 175, 177).

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990) (cited on page 149).

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “Address-Sanitizer: A Fast Address Sanity Checker”. In: *USENIX ATC*. 2012 (cited on pages 151, 167, 175).

[Fio+22] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. “LibAFL: A Framework to Build Modular and Reusable Fuzzers”. In: *CCS*. ACM, 2022 (cited on pages 151, 155, 172).

[Kle+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating Fuzz Testing”. In: *CCS*. ACM, 2018 (cited on pages 152, 178).

[PBR20] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. “AFLNET: A Greybox Fuzzer for Network Protocols”. In: *ICST*. IEEE, 2020 (cited on pages 156, 175).

[boof] *boofuzz: Network Protocol Fuzzing for Humans*. https://boofuzz.readthedocs.io/ (cited on page 156).

[fuzzw] *Fuzzowski Network Fuzzer*. https://github.com/nccgroup/fuzzowski (cited on page 156).

[CVE14-0160] *CVE-2014-0160*. https://www.cve.org/CVERecord?id=CVE-2014-0160 (cited on page 156).

[Cha+17] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. “SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations”. In: *S&P*. IEEE, 2017 (cited on page 156).

[Yah+21] Moosa Yahyazadeh, Sze Yiu Chau, Li Li, Man Hong Hue, Joyanta Debnath, Sheung Chiu Ip, Chun Ngai Li, Endadul Hoque, and Omar Chowdhury. “Morpheus: Bringing The (PKCS) One To Meet the Oracle”. In: *CCS*. 2021 (cited on page 156).

[Asc+19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. “NAUTILUS: Fishing for Deep Bugs with Grammars”. In: *NDSS*. The Internet Society, 2019 (cited on page 156).

[Goo] Google. *honggfuzz*. https://honggfuzz.dev/ (cited on pages 156, 175).

[Lee+14] Hyojeong Lee, Jeff Seibert, Md. Endadul Hoque, Charles Killian, and Cristina Nita-Rotaru. “Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations”. In: *ICDCS*. 2014 (cited on page 156).

[Hoq+16] Endadul Hoque, Hyojeong Lee, Rahul Potharaju, Charles Killian, and Cristina Nita-Rotaru. “Automated Adversarial Testing of Unmodified Wireless Routing Implementations”. In: *IEEE/ACM Trans. Netw.* 24.6 (Dec. 2016) (cited on page 156).

[Mae+22] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. “TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries”. In: *USENIX Security Symposium*. USENIX Association, 2022 (cited on pages 156, 178).

[Som16] Juraj Somorovsky. “Systematic Fuzzing and Testing of TLS Libraries”. In: *CCS*. 2016 (cited on pages 156, 179).

[Kar+23] Imtiaz Karim, Abdullah Al Ishtiaq, Syed Rafiul Hussain, and Elisa Bertino. “BLEDiff: Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations”. In: *S&P*. IEEE, 2023 (cited on pages 156, 157).

[Nat22] Roberto Natella. “StateAFL: Greybox Fuzzing for Stateful Network Servers”. In: *Empirical Softw. Engg.* 27.7 (Dec. 2022) (cited on pages 156, 175).

[SCP14] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. “SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications”. In: *IEEE Trans. Softw. Eng.* 40.7 (July 2014) (cited on pages 156, 157).

[Ban+06] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. “SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr”. In: *ISC*. Lecture Notes in Computer Science. Springer, 2006 (cited on page 157).

[Hus+21] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. “Noncompliance as Deviant Behavior: An Automated Black-Box Noncompliance Checker for 4G LTE Cellular Devices”. In: *CCS*. 2021 (cited on page 157).

[PRS15] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. “Protocol State Machines and Session Languages: Specification, implementation, and Security Flaws”. In: *S&P Workshops*. IEEE Computer Society, 2015 (cited on page 157).

[Hoq+17] Md. Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. “Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs”. In: *DSN*. 2017 (cited on page 157).

[Sch+22] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. “Nyx-Net: Network Fuzzing with Incremental Snapshots”. In: *EuroSys*. 2022 (cited on page 157).

[DPR18] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. “Inferring OpenVPN State Machines Using Protocol State Fuzzing”. In: *EuroS&P Workshops*. 2018 (cited on page 157).

[ME04] Madanlal Musuvathi and Dawson R. Engler. “Model Checking Large Network Protocol Implementations”. In: *NSDI*. 2004 (cited on page 157).

[Del+17] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. “Implementing and Proving the TLS 1.3 Record Layer”. In: *S&P*. IEEE, 2017 (cited on page 157).

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, et al. “The dogged pursuit of bug-free C programs: the Frama-C software analysis platform”. In: *Commun. ACM* 64.8 (2021) (cited on page 157).

[Ler09] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009) (cited on page 157).

[CK14] Véronique Cortier and Steve Kremer. “Formal Models and Techniques for Analyzing Security Protocols: A Tutorial”. In: *Found. Trends Program. Lang.* 1.3 (2014) (cited on pages 158, 159).

[AHK23] Max Ammann, Lucca Hirschi, and Steve Kremer. *DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing*. Cryptology ePrint Archive, Paper 2023/057. https://eprint.iacr.org/2023/057. 2023. URL: https://eprint.iacr.org/2023/057 (cited on pages 165, 166, 172, 173, 176).

[HHP20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. “Magma: A Ground-Truth Fuzzing Benchmark”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (2020) (cited on page 174).

[CVE16-2108] *CVE-2016-2108*. https://www.cve.org/CVERecord?id=CVE-2016-2108 (cited on page 175).

[aflnwe] *aflnwe*. https://github.com/thuanpv/aflnwe (cited on page 175).

[NP21] Roberto Natella and Van-Thuan Pham. “ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing”. In: *ISSTA*. 2021 (cited on page 175).

[Ser17] Kostya Serebryany. *OSS-Fuzz: Google’s Continuous Fuzzing Service for Open Source Software*. USENIX Security Symposium (talk). 2017 (cited on page 175).

[Wol] WolfSSL Project. *Fuzz Testing*. https://www.wolfssl.com/fuzz-testing/ (cited on page 175).

[WA24] Johannes Wilson and Mikael Asplund. “Analysing TLS Implementations Using Full-Message Symbolic Execution”. In: *NordSec*. Lecture Notes in Computer Science. Springer, 2024 (cited on page 181).

[GHK26] Tom Gouville, Lucca Hirschi, and Steve Kremer. *DDYF: Differential Dolev-Yao Fuzzing of Cryptographic Protocols (eprint v1.0)*. https://eprint.iacr.org/archive/2026/991/20260519:065203. 2026 (cited on page 182).

[Böh18] Marcel Böhme. “STADS: Software Testing as Species Discovery”. In: *ACM Trans. Softw. Eng. Methodol.* 27.2 (2018) (cited on page 186).

[BLW21] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. “Estimating Residual Risk in Greybox Fuzzing”. In: *ESEC/SIGSOFT FSE*. ACM, 2021 (cited on page 186).

[Kim+25] Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R. Jeong, et al. *ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System*. arXiv:2509.14589. 2025 (cited on page 187).

[Jac+19] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. “Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures”. In: *CCS*. ACM, 2019 (cited on page 187).

[CJ19] Cas Cremers and Dennis Jackson. “Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman”. In: *CSF*. IEEE, 2019 (cited on page 187).

[CJL23] Cas Cremers, Charlie Jacomme, and Philip Lukert. “Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis”. In: *CSF*. IEEE, 2023 (cited on page 187).

[BHM21] Xavier Boyen, Thomas Haines, and Johannes Müller. "Epoque: Practical End-to-End Verifiable Post-Quantum-Secure E-Voting". In: *EuroS&P*. IEEE, 2021 (cited on page 191).

[Asc+20] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "IJON: Exploring Deep State Spaces via Fuzzing". In: *S&P*. IEEE, 2020 (cited on page 192).

[Ba+22] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. "Stateful Greybox Fuzzing". In: *USENIX Security Symposium*. USENIX Association, 2022 (cited on page 192).

[AC06] Martín Abadi and Véronique Cortier. "Deciding knowledge in security protocols under equational theories". In: *Theor. Comput. Sci.* 367.1-2 (2006) (cited on page 192).

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ESEC/SIGSOFT FSE*. ACM, 2005 (cited on page 196).

[CD09] Sagar Chaki and Anupam Datta. "ASPIER: An Automated Framework for Verifying Security Protocol Implementations". In: *CSF*. IEEE, 2009 (cited on page 196).

[Cla+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003) (cited on page 197).

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *CAV*. LNCS. Springer, 2000 (cited on page 197).

[Bon+20] Xavier Bonnetain, Anne Canteaut, Véronique Cortier, Pierrick Gaudry, Lucca Hirschi, Steve Kremer, Stéphanie Lacour, Matthieu Lequesne, Gaëtan Leurent, Léo Perrin, et al. *Le traçage anonyme, dangereux oxymore (Anonymous Tracing: A Dangerous Oxymoron)*. HAL report hal-02997228. 2020. URL: https://hal.inria.fr/hal-02997228 (cited on page 197).