A Generic Security API for Symmetric Key Management on Cryptographic Devices

Véronique Cortier[a], Graham Steel[b]

[a] *CNRS, Loria, UMR 7503, Vandœuvre-lès-Nancy, F-54500 France* cortier@loria.fr
[b] *INRIA Project ProSecCo, 23 Avenue d'Italie, 75013 Paris, France*

Abstract

We present the design of a new symmetric key management API for cryptographic devices intended to implement security protocols in distributed systems. Our API has a formal security policy expressed in terms of invariants under various threat scenarios, and proofs of security in the symbolic model. This sets it apart from previous APIs such as RSA PKCS#11, which are under-specified, lack a clear secuity policy and are often subject to attacks.

Our design is based on the principle of explicitness: the security policy for a key must be given at creation time, and this policy is then included in any ciphertext containing the key. The policy is expressed in terms of a position in a hierarchy of keys and a set of agents. Our API also contains novel features such as the possibility of insisting on a freshness check before accepting an encrypted key for import. To show the applicability of our design, we give an algorithm for automatically instantiating the API commands for a given key management protocol. We demonstrate the algorithm on a set of symmetric key establishment protocols from the Clark-Jacob suite. We show that in the restricted mode of operation where freshness checks are required, some protocols from the test suite cannot be implemented: precisely those now known to be susceptible to replay attacks.

This paper is an extended version of a paper published at the ESORICS conference in September 2009. It contains proofs of more fine-grained security properties than the original paper (for the same API), in particular in the case where some but not all long-term keys on a particular token are lost to the attacker. Since the original paper was submitted, another key management API with a security proof has appeared in the literature due to Cachin and Chandran [4]. This present paper contains a comparison of the two designs and their security properties, as well as a more detailed comparison to other API designs.

Keywords: Security APIs, key management, PKCS#11, cryptographic devices
2000 MSC: 94A60

1. Introduction

Distributed systems designed to operate in insecure environments are increasingly making use of tamper-resistant cryptographic devices such as smartcards and hardware security modules (HSMs) to implement the endpoints of their protocols. These devices must offer

an application program interface (API) to their host machines that allows the management and use of the keys. Since the host machine might in the worst case be executing malicious code, this interface must not only allow access to cryptographic functionality, but also enforce a policy, i.e. no matter what sequence of commands are called, some security properties, such as the secrecy of sensitive keys, continue to hold.

The ability of these APIs to enforce their policies has been the subject of formal and informal analysis in recent years. Open standards such as RSA PKCS#11 [20] and proprietary solutions such as IBM's Common Cryptographic Architecture [6] have been shown to have flaws which may lead to serious attacks [2, 3, 8, 9, 12, 17]. The situation is exacerbated by the fact that these APIs lack a clearly specified security policy, as highlighted in disputes over what does and does not constitute an attack [14]. All this leaves the application developer in a very difficult position.

The first contribution of this paper is the design of a new API for key management on cryptographic devices. Novel features of our proposal include an explicit tagging scheme for encryptions which makes it possible to enforce a security policy, and an option to require freshness checks on imported keys. Our scheme explicitly identifies agents for which cryptographic material is supposed to be used. This is used in the policy: for example, our API will encrypt data only if the agents that are granted access to the encryption key are all also granted access to the data. Our design follows the well-known principle of explicitness. A novel feature is the transportation of the security policy in all encrypted messages.

The second contribution is the definition of a formal threat model and security policy for the API, together with proofs of security. Our specifications are based on invariants. Given different threat scenarios in terms of compromised agents and keys, stronger or weaker invariants are shown to hold. Our proofs are in an abstract model of cryptography similar to those used in protocol analysis.

The third contribution is an algorithm showing how to use our API to implement a symmetric key management protocol, which we demonstrate on all the protocols of a relevant class in the Clark-Jacob library, to show the applicability of our design.

Related Work. Flaws in key management APIs were first found by Longley and Rigby [17] and then by Bond and Anderson [2] and Clulow [8]. The use of formal methods by Cortier, Keighren and Steel [9], Delaune, Kremer and Steel [12] and Bortolozzo, Centenaro, Focardi and Steel [3] lead to the discovery of another wave of vulnerabilities. There have been fewer results on proving APIs to be secure. Courant gave a proof, partially mechanised in Coq, of the security of a variant of the IBM CCA API [11]. Cortier, Keighren and Steel showed the security of another variant of the CCA [9]. Both these proofs assume a bounded number of fresh keys. Fröschle and Steel showed how to use abstractions to prove security for an unbounded number of fresh keys, but this was for a very limited fragment of PKCS#11 [13]. Our proofs in this paper are for unbounded fresh keys and for a much richer API. Since the original version of this paper was published at ESORICS in September 2009 [10], another proposal for a key management interface was published by Cachin and Chandran [4]. We give a detailed comparison between our design and theirs in section 2.

Paper Outline. In section 2 we present our API informally and compare it to other designs. We give our formal model and a formal definition of the API in section 3. In section 4, we explain our formal modelling of the threat scenario and give the security properties for which proofs appear in Appendix A. We demonstrate our algorithm for instantiating the API for a symmetric-key protocol in section 5, giving results on the Clark-Jacob suite. Finally we give conclusions and further work in section 6.

2. Informal Presentation of the Generic API

In this section, we present the ideas behind our API design and compare it to other proposals. Formal definitions will follow in the next section.

Handles. We assume a tamper resistant device (TRD) with a limited (but for the moment unspecified) amount of memory, capable of symmetric key cryptography. The device is to be deployed to facilitate the execution of symmetric key distribution protocols, and the subsequent use of the session keys established by these protocols. A user should never have direct access to the stored secret values but should use the API commands to instruct the TRD to encrypt and decrypt for him, referring to the secrets by their *handles*, which are like pointers or names and are a standard feature of APIs like PKCS#11 [20]. The value of a handle in general reveals nothing about the value it refers to.

Attributes. Every object (key, nonce) stored on the device has two *attributes* or pieces of metadata stored along with it: the first is its *level*, which may be 0 (public value), 1 (secret non-key value), 2 (secret *session key*) or 3 (secret *long term key*). By *session key* we mean a key intended for encrypting and decrypting data: these keys are called *data keys* in IBM's CCA API [6] and *encrypt* and *decrypt* keys in PKCS#11 [20]. By *long term key* we mean a key intended primarily for encrypting and decrypting other keys. Such keys are called *key encryption keys* or KEKs in the CCA and *wrap* and *unwrap* keys in PKCS#11.

The second attribute is a *set of agents* who have access to the object. These attributes correspond to the (informal) specification given in the NIST standard FIPS 140-2 [15, §4.7], but note they could quite easily be extended without requiring fundamental changes to our API or proofs: one could extend the key hierarchy with some higher level keys to accommodate protocols requiring more levels such as Kerberos [18], or associate a group or process identifier to an object instead of a set of agents. The important point is that these attributes provide an ordering, which is what we will use to define and enforce a security policy. The encryption command will only allow keys (objects of level 2 or 3) to encrypt objects of a strictly lower level. Furthermore, if the encryption key is associated with a set of users S then all the objects encrypted must be associated to sets S' such that S is a subset of S'. In other words, if a secret object is associated to a group of users S', then it should not be made available to any users outside that set by encrypting it under a key they have access to.

Tagging scheme. The encryption command takes two parameters: a handle for the encryption key, and a list of objects to be encrypted. It then builds an encryption containing explicit *tags* identifying the objects and their associated attributes. This tagging scheme allows the objects to be treated correctly on decryption: public objects can be returned to the calling program, whereas encrypted keys will be stored on the device along with their correct metadata.

Freshness checks. An unusual feature of our API is the ability to make freshness checks on imported keys. If the API is running in its restricted mode, where freshness checks are required, then every time the decryption command is called on an encrypted packet that contains a session key, the command will only succeed if a nonce is also contained in the encrypted package. Furthermore, this nonce must have been generated by the TRD undertaking the decryption, and must not have been used for a previous freshness test. This is of course not the only way to guarantee freshness, but it turns out that this simple test gives us robust security properties under compromised keys (see section 4), as well as allowing us to implement all the symmetric key management protocols from the Clark-Jacob protocol suite that use nonces (as opposed to timestamps) for freshness and are not susceptible to replay attacks (see section 5).

Initialisation. Our API does not contain explicit initialisation commands for loading long-term keys. We assume that this can be performed in a secure environment by some kind of *personalisation* process, as it is referred to in industry.

2.1. Comparison with Existing APIs

The most widely used API for TRDs is the RSA standard PKCS#11, also known as 'Cryptoki' [20]. PKCS#11-based APIs have been shown to be vulnerable to a variety of attacks whereby sensitive keys are compromised [3, 8, 12]. Our API has several features designed specifically to counter these kinds of threats. Firstly, we insist on an encryption scheme whereby data from the host machine and secret data from inside the TRD are tagged differently when encrypted to avoid confusion. PKCS#11 does not do this, and this confusion is exploited by many of the known attacks, e.g. by using the decryption function of the API to reveal an encrypted key in clear. Secondly we insist that keys are stored with specific roles, either as session keys or long term keys, and these roles cannot be changed. Allowing the roles of keys to change (signified by their attributes *encrypt*, *decrypt*, *wrap* and *unwrap* in PKCS#11) is another major source of vulnerabilities in the Cryptoki API. We store the identities of agents for whom a key is intended to be used inside the TRD, and include these identities as tags in our encryption scheme. PKCS#11 makes no such provision: it supports only two user profiles, a normal user and a security officer who cannot use keys but can e.g. reset the login PIN. Finally, we note that PKCS#11 makes no provision for freshness checks: this would all have to be carried out in client code running on the possibly insecure host machine.

IBM's CCA API [6] was designed for a previous generation of TRDs that had limited memory and yet needed to use many different keys. These 'working keys' were stored on

the host machine encrypted under a 'master key' unique to the device. To identify different types of working keys, different 'control vectors' were XORed against the master key before encrypting or decrypting the working key. This use of XOR was found to be problematic in the attacks found by Bond [2]. Nonetheless fragments of the API have been shown secure by Cortier, Keighren and Steel [9]. However, this design again contains no support (within the TRD) for multiple users, and no freshness tests.

A recent article by Cachin and Chandran of IBM research, published since the first version of our paper was submitted, proposes a new API for the next generation of IBM products [4]. This API is designed for a rather different scenario from ours: a single TRD is assumed, with a large memory (e.g. a central key server for an organisation). Multiple users are explicitly supported: each stored key has associated access control rules for each user. Instead of requiring keys to be given a type, keys are assigned a type based on history. For example, when a key is used for the first time for wrapping, it becomes a key of wrap/unwrap type, and can no longer be used for any other operation such as encryption or decryption. Furthermore, dependencies between keys created by e.g. wrapping one key under another are explicitly tracked with tables of ancestors and dependents. These tables are used to enforce a security policy. For example, if key k has been read by user a, then a can only wrap key k' under k if he also has rights to read key k', since the wrapping operation will reveal the key to him.

Cachin and Chandran's API does not include provision for making freshness checks, but in their restricted scenario where keys are not to be shared between devices, these do not seem necessary. Their security properties are weaker than ours in one sense, in that they do not allow key corruption, but stronger in another, in that their proofs are in the cryptographic style where the attacker is an arbitrary polynomial time probabilistic Turing machine, whereas ours is restricted to a certain set of operation in the abstract model. However, their assumptions on the encryption and decryption algorithms are standard (indistinguishability under adaptive chosen ciphertext attack, or IND-CCA2) and it seems likely that though there is not yet a 'soundness' result suitable for symbolic models of key management APIs, showing that proofs in the symbolic model correspond to proofs in the cryptographic model, such a proof under standard assumptions could be made for our API (cryptographically awkward constructs such as key cycles are avoided by the hierarchy on key types). The Cachin and Chandran API also includes some operations not present in our API, for example encryption and decryption (but not wrapping and unwrapping) using asymmetric keys. We feel these could easily be added to our design if necessary.

The essential differences between the designs are in the key usage policy, and stem from the different scenarios in which the interfaces are expected to be deployed: we expect memory on our devices such as smartcards, SIM cards, USB keys etc. to be relatively restricted, hence we cannot store history of operations as Cachin and Chandran do. Furthermore, we expect the same keys to be stored on several different devices with no shared memory, hence we cannot rely on an up-to-date log. This constrains us to demand that the usage policies for the keys, given by the key level and the set of agents, are given at key creation time and are immutable. Cachin and Chandran assume a single server with a relatively large secure memory which is the only place where keys will reside, hence they can base

their security on the usage history of a key offering more flexibility. It seems clear that a complete solution will require some limited resource devices and some larger servers, hence combining our ideas would be an interesting future project. We comment further on this in section 6.

3. Formal Model and description of the API

We present our formal model in the first two subsections. We then formally describe the behavior of our API within our model. The reader keen to see the description of the API with no interest in formal models may however go directly to Section 3.3 and skip the formal definitions provided in Figure 1.

3.1. Syntax

Messages are represented using a term algebra. We assume a finite set of agents Agent and countably infinite sets of nonces Nonce, keys Key, and other atoms Text. We also assume a countably infinite set of variables Var, among which we distinguish a set VarKey of variables of sort key and a set $\mathsf{VarNonce}$ of sort nonce.

$$\begin{array}{rcl} \mathsf{Keyv} & ::= & \mathsf{Key} \mid \mathsf{VarKey} \\ \mathsf{Noncev} & ::= & \mathsf{Nonce} \mid \mathsf{VarNonce} \\ \mathsf{Msg} & ::= & \mathsf{Agent} \mid \mathsf{Text} \mid \mathsf{Var} \mid \{\mathsf{Msg}\}_{\mathsf{Keyv}} \mid \langle \mathsf{Msg}, \mathsf{Msg} \rangle \\ \mathsf{Handle} & ::= & h_a^\alpha(\mathsf{Nonce}, \mathsf{Msg}, i, S) \end{array}$$

where $i \in \{0, 1, 2, 3\}, S \subseteq \mathsf{Agent}, a \in \mathsf{Agent}, \alpha \in \{r, g\}$. The set Keyv denotes both keys and key variables. Similarly, the set Noncev denotes both nonces and nonce variables. Terms of sort Msg are built from Keyv, Noncev, Text and Agent using encryption and pairing. The term $\{m\}_k$ represents the message m encrypted by the key k while the term $\langle m_1, m_2 \rangle$ represents the concatenation of the two messages m_1 and m_2. We may write $t_1, t_2, \ldots, t_n$ instead of $\langle t_1, \langle t_2, \langle \ldots, t_n \rangle \ldots \rangle \rangle$. A *substitution* $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a function that replaces variables by terms. The application of a substitution θ to a term t is denoted by $t\theta$. The *positions* of term t are defined as usual and $t|_p$ denotes the term at position p in t. For example, the set of positions of the term $t = \{\langle N_1, N_2 \rangle\}_K$ is $\{\epsilon, 1, 1.1, 1.2, 2\}$ where ϵ denotes the root position and $t|_{1.2} = N_2$.

The API does not give direct access to secret messages but provides the user with a handle that can be used later to indicate to the API to use a specific message. A handle $h_a^\alpha(n, m, i, S)$ represents a reference for a message m of security level i stored on the TRD belonging to a (each TRD is associated with a single user). Note that in practice, a handle is typically a pointer to a key and its attributes. Here, we use a composed term to represent handles in order to make explicit the memory contents pointed to by the handle. The nonce n is a special kind of nonce that we will call a 'handle nonce' that appears only in the first argument of h terms, and serves in the model to avoid confusion between different handles that refer to the same data. The set S represents the set of users that are allowed to access to m. By convention, the special constant All will indicate public data.

The label α distinguishes the handles corresponding to values m generated by the API ($\alpha = g$) from values m received by the API ($\alpha = r$). This distinction allows the API to check for freshness. The values stored inside the TRD will typically be nonces or keys. However, in order to reflect the inability of an TRD to check whether an arbitrary bitstring is a key or not, we *a priori* allow any message to be stored inside the TRD. As explained in Section 2, the security level i can take four values:

- 0: public data (e.g. a public nonce used for freshness purpose)
- 1: secret data that are not used for encryption (typically nonces)
- 2: session keys
- 3: long term keys

Note that the security level represents the intended level of security. The goal of the API is to ensure that (typically) secret data are not revealed. In particular, public data should be accessible to anyone. In what follows, we may write $m, 0$ instead of $m, 0, \mathsf{All}$, to indicate that m is of level 0 and accessible to all users.

Terms of sort Msg are called messages while terms of sort Handle are called handles. We denote by $\{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$ the substitution σ that replaces the variable x_i with the term t_i. σ is well-sorted if t_i is of the same sort as x_i. The *domain* of σ, denoted by $dom(\sigma)$ is the set $\{x_1, \dots, x_k\}$. The application of a substitution σ to a term t is denoted by $t\sigma$. In what follows, we only consider well-sorted substitution.

We consider the set $\mathcal{P} = \{P_a \mid a \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ of predicates. P_a with $a \in \mathsf{Agent}$ represents the knowledge of an agent a. The predicate P_{int} is a special predicate that represents the knowledge of the attacker.

3.2. Model

Our model is a state-based transition system.

A rule that describes a possible evolution of the system is an expression of the form

$$P_{i_1}(u_1), \dots, P_{i_k}(u_k) \overset{N_1,\dots,N_p}{\Longrightarrow} P_{j_1}(v_1), \dots, P_{j_l}(v_l)$$

where the u_i, v_i are messages or handles possibly with variables, the N_i are variables and $i_1, \dots, i_k,\ j_1, \dots, j_l$ belong to $\mathsf{Agent} \cup \{\mathsf{int}\}$. Such a rule represents the fact that when the API i_p are given u_p then the API j_p output v_p. Examples of API rules are given in Section 3.4.

An important example of rules is the following set $\mathsf{ATTACKER}$ of rules that represents the ability of an attacker to pair and project and to encrypt and decrypt when he knows the key.

$$
\begin{array}{rcll}
P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(\langle x, y\rangle) & \text{Concatenation} \\
P_{\mathsf{int}}(\langle x, y\rangle) & \Rightarrow & P_{\mathsf{int}}(x) & \text{Projection} \\
P_{\mathsf{int}}(\langle x, y\rangle) & \Rightarrow & P_{\mathsf{int}}(y) & \text{Projection} \\
P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(\{x\}_y) & \text{Encryption} \\
P_{\mathsf{int}}(\{x\}_y), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(x) & \text{Decryption}
\end{array}
\qquad (\mathsf{INTRUDER})
$$

A state of our execution model is the current knowledge of the attacker and the users. It is formally represented by a family $\{S_b \mid b \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ where int is a special index representing the attacker. The S_b are sets of messages and handles. Given a family $\mathcal{S}$ of sets and an index $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$, we denote by $\mathcal{S}_b$ the set S_b of $\mathcal{S}$ indexed by b. The global state of the system evolves following the rules. Given a set of rules $\mathcal{R}$, we say that a state $\mathcal{S}$ is accessible in one step from a state $\mathcal{S}'$, denoted by $\mathcal{S} \Rightarrow_{\mathcal{R}} \mathcal{S}'$ if there exists a rule of the form $P_{a_1}(u_1), \ldots, P_{a_k}(u_k) \overset{N_1,\ldots,N_p}{\Longrightarrow} P_{b_1}(v_1), \ldots, P_{b_l}(v_l)$ of $\mathcal{R}$ and a substitution θ such that

- $u_i\theta \in \mathcal{S}_{a_i}$ for any $1 \leq i \leq k$;
- $N_j\theta$ are fresh nonces (that do not appear in $\mathcal{S}$);
- $\mathcal{S}'$ is the smallest family such that $\mathcal{S}_b \subseteq \mathcal{S}'_b$ for any $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$ and $v_i\theta \in \mathcal{S}'_{b_i}$ for any $1 \leq i \leq l$.

$\Rightarrow^*_{\mathcal{R}}$ denotes the reflexive and transitive closure of $\Rightarrow_{\mathcal{R}}$. We may omit $\mathcal{R}$ when the set of rules is clear from the context. Note that the arrow $\Rightarrow$ is used here in two different contexts with different meanings: between two states, $\mathcal{S} \Rightarrow \mathcal{S}'$ denotes a transition, while in the intruder and protocol rules, it is part of the syntax to describe the rules.

Interestingly, we retrieve the usual deducibility notion (*i.e.* what an attacker can compute from a set of messages) by saying that a term m is deducible from a set of terms S, which is denoted by $S \vdash m$, whenever there exists $\mathcal{S}'$ such that $\mathcal{S} \Rightarrow^*_{\mathsf{ATTACKER}} \mathcal{S}'$ and $m \in \mathcal{S}'_{\mathsf{int}}$ where $\mathcal{S}$ is defined by $\mathcal{S}_a = \emptyset$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}_{\mathsf{int}} = S$.

3.3. Formal Definition of the API

Our generic API is fully specified by the family of rules given in Figure 1 using the language of our formal model defined in section 3.2. The set of all the rules is denoted by API. We explain the three kinds of commands of our API with informal pseudocode.

Secure and Public Generate. The API allows a user (e.g. an agent acting in a protocol) to generate a new secret nonce or key of security level $i \in \{1, 2\}$ for a group $S \subseteq \mathsf{Agent}$ of agents. The level and group are given as inputs to the command. Concretely, it generates both a nonce or a key K and a handle, it returns only the handle to the user and binds the handle to the newly generated object K, the key metadata i and S, and to the Boolean flag g that indicates that the data K has been generated locally.

Additionally, the API allows the user to generate a public value of security level 0, in which case it returns both a handle to the object and the value itself.

$$\overset{N,K}{\Rightarrow} P_a(h^g_a(N,K,i,S)) \quad i \in \{1,2\} \qquad \textbf{(Secure Generate)}$$

$$\overset{N,K}{\Rightarrow} P_a(K), P_a(h^g_a(N,K,0,\mathsf{All})) \qquad \textbf{(Public Generate)}$$

where $N \in \mathsf{Noncev}$ and $K \in \mathsf{Noncev}$ if $i = 1$, $K \in \mathsf{Keyv}$ otherwise.

$$P_a(h^\alpha_a(N,K,i_0,S_0)), P_a(m_1), \dots, P_a(m_p) \Rightarrow P_a(\{m'_1, \dots, m'_p\}_K) \qquad \textbf{(Encrypt)}$$

where

- $\alpha \in \{r, g\}$, $p \in \mathbb{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2,3\}$, $N \in \mathsf{Noncev}$, $K \in \mathsf{VarKey}$;
- $m'_j = m_j, 0, \mathsf{All}$ if $m_j \in \mathsf{Var}$ is a variable.
- $m'_j = K_j, i_j, S_j$ with $i_j < i_0$ and $S_0 \subseteq S_j$ if $m_j \in \mathsf{Handle}$ is a handle of the form $h^{\alpha_j}_a(N_j, K_j, i_j, S_j)$ with $N_j \in \mathsf{Noncev}$ and $K_j \in \mathsf{VarKey}$.

$$P_a(h^\alpha_a(N,K,i_0,S_0)), P_a(\{m_1, \dots, m_p\}_K), \bigcup_{j \in L} P_a(m'_j) \overset{N_1,\dots,N_p}{\Rightarrow} \bigcup_{j \notin L} P_a(m'_j)$$

(Decrypt)

where

- $L \subseteq \{1, \dots, p\}$, $\alpha \in \{r, g\}$, $p \in \mathbb{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2,3\}$, $K \in \mathsf{VarKey}$, $N, N_1, \dots, N_p \in \mathsf{VarNonce}$;
- for any $j \in L$, $m'_j = h^g_a(N_j, X_j, 0, \mathsf{All})$ if m_j is of the form $X_j, 0, \mathsf{All}$ and $m'_j = h^g_a(N_j, X_j, i_j, S_j)$ if m_j is of the form i_j, S_j, X_j with $i_j \geq 1$ and $X_j \in \mathsf{Var}$, $N_j \in \mathsf{Noncev}$.
- for any $j \notin L$, $m'_j = X_j$ if m_j is of the form $X_j, 0, \mathsf{All}$ (data of security level 0 are given to the user), $X_j \in \mathsf{Var}$ and $m'_j = h^r_a(N_j, K_j, i_j, S_j)$ if m_j is of the form K_j, i_j, S_j with $N_j \in \mathsf{Noncev}$, $K_j \in \mathsf{VarKey}$, $i_j \geq 1$, $i_j < i_0$ and $S_0 \subseteq S_j$.

Note that the rules are parametrized by sets $S, S_0, S_1, \dots, S_n$ which can be any subset of Agent.

Figure 1: Formal Description of API rules

Given a random generation function $\mathsf{RGenerate}$, and input i, S, the algorithm functions as follows (if $i = 0$ then S is automatically set to All)

1. $K \leftarrow \mathsf{RGenerate}$, $N \leftarrow \mathsf{RGenerate}$
2. store $N \mapsto (K, i, S, TRUE)$
3. if $i = 0$ return N, K else return N

The idea of storing a public value securely on the TRD is that this will allow the API to ensure it is only used once to guarantee freshness of session keys. This is particularly important when implementing protocols where a fresh session key received inside an encrypted packet should be accepted only if the packet also contains a (fresh) nonce sent at a previous step of the protocol. We will see these freshness tests indeed allow to securely implement protocols in section 4.3, offering a protection against replay attacks.

Encrypt. The encryption command allows the user to encrypt a mixture of public data, level 1 data (i.e. nonces or other secrets), and level 2 data (session keys). This the user does by first supplying the handle for the encryption key, and then for all the objects to be encrypted $m_1, \dots m_n$, the user supplies either the handle or (in the case of level 0 data) the value itself. All encrypted data must be of a strictly lower security level than the encryption key used, meaning that session keys (level 2) can only be encrypted under long-term keys (level 3). Additionally, for every object stored on the TRD that will be included in the encrypted payload, the set S of agents associated with that object must be a superset of the set of agents associated with the encryption key. The rationale is that by encrypting the data under a key associated with a list of agents S', we are effectively making that data available to those agents, so we should only make data available to agents who are supposed to have access to it. In the encrypted packet created by the API, each piece of data to be encrypted is tagged with both its security level, and with the list of agents allowed access to it. The API then returns the encrypted value.

Given input N and plaintexts $m_1, \dots, m_n$, the algorithm functions as follows

1. $(K, i, S, _) \leftarrow N$ % the API retrieves the data associated to N
2. For each m_j, case m_j of
   $(K', i', S', _) \leftarrow m_j$: % m_j refers to some data stored on the TRD
   if $(i' < i)$ and $(S \subseteq S')$ then $m'_j \leftarrow (K', i', S')$ else **break**
   $m'_j \leftarrow (m_j, 0, \mathsf{All})$ otherwise
3. return $\mathsf{encrypt}(K; m'_1, \dots, m'_n)$ % the m'_j are encrypted with K.

Decrypt. On receiving a ciphertext, a user can call the decrypt function to have it deciphered. The user supplies the encrypted packet and the handle for the decryption key. The API works through the packet. For each decrypted object $m_1, \dots, m_n$, the API examines the tags first and makes the same checks as are made for encryption: the level i must be strictly lower than the security level of the key, and the set S must be a superset of the set S' associated with the decryption key. It may seem redundant to repeat the checks which were made on encryption, but it is vital to limit the loss of security in the event of a key becoming compromised. For each object the API returns either a handle to the object which is now stored on the TRD (when the object is level 1 or 2), or the value of the object itself (when it is of level 0).

There is one further detail to the decryption rule. The user can give a set of indexes L and corresponding handles for each index. These handles should point to objects of security level 1 or 0 and are to be used as test values. Note that the test handles must have the g superscript, i.e. they must point to values generated on the device receiving the

keys. The set L is treated as the empty set when not provided by the user. The decryption command will then only succeed if all of these tests are passed, i.e. values identical to those pointed to by the test handles are found in the ciphertext. This testing behaviour is vital for avoiding replay attacks, as we will show in section 4.3.

Given input N, L and ciphertext C, the algorithm functions as follows. Note that the **abort** instruction breaks out of the whole algorithm, not just the current step, and returns a single identical error message in all cases.

1. $(K, i, S, _) \leftarrow N$ % the API retrieves the data associated to N
2. if $(i > 1)$ $m_1, \ldots, m_n \leftarrow \mathsf{decrypt}(K; C)$ else **abort**
3. For each m_j case m_j of
   - $(X, 0, \mathsf{All})$ and $(j, N') \in L$: % the public value X has to be checked for equality
     - (a) $(K', i'', S'', B) \leftarrow N'$
     - (b) if $i'' \neq 0$ or $S'' \neq \mathsf{All}$ or $B \neq TRUE$ or $X \neq K'$ **abort**
     - (c) $m'_j \leftarrow \emptyset$
   - $(X, 0, \mathsf{All})$: $m'_j \leftarrow X$ % the public value X is simply decrypted
   - (X, i', S') and $(j, N') \in L$: % the private value X has to be checked for equality
     - (a) $(K', i'', S'', B) \leftarrow N'$
     - (b) if $i'' \neq i'$ or $S'' \neq S'$ or $B \neq TRUE$ or $X \neq K'$ **abort**
     - (c) $m'_j \leftarrow \emptyset$
   - $(X, i', S', _)$: % the private value X has to be decrypted and stored on the TRD
     if $(i' < i)$ and $(S \subseteq S')$ then
     - (a) $N' \leftarrow \mathsf{RGenerate}$
     - (b) store $N' \mapsto (X, i', S', FALSE)$
     - (c) $m'_j \leftarrow N'$

     else **break**
4. return $(m'_1, \ldots, m'_j)$

Note that our API never performs explicit nested encryption nor decryption in a single command. However, our API does allow a message to be sent that contains nested encryption. This occurs, for example, when the **Encrypt** command is called on a public message (of level 0) that is already a ciphertext.

3.4. Using the API

We now show how the API's commands can be used to implement a simple protocol from the literature, the Carlsen's Secret Key Initiator Protocol [5, Figure 2]

1.	A	$\rightarrow$	B	:	A, Na
2.	B	$\rightarrow$	S	:	A, Na, B, Nb
3.	S	$\rightarrow$	B	:	$\{K_{ab}, N_b, A\}_{K_{bs}}, \{N_a, B, K_{ab}\}_{K_{as}}$
4.	B	$\rightarrow$	A	:	$\{N_a, B, K_{ab}\}_{K_{as}}, \{N_a\}_{K_{ab}}, N'_b$
5.	A	$\rightarrow$	B	:	$\{N'_b\}_{Kab}$

The aim of the protocol is to establish a fresh session key K_{ab} (that is, of security level 2) for agents A and B using a key server S. In the first message, A sends her name and a fresh nonce to B. In message 2, B forwards these values together with his own fresh nonce to the server S. The server generates K_{ab} and encrypts it first for B, under B's long term key K_{bs}, in a packet together with his nonce and A's name, and then for A, under her long term key K_{as}, together with her nonce and B's name. The server sends both packets to B. In message 4, B forwards to A her encrypted packet, A's nonce N_a encrypted under the session key K_{ab}, and a further fresh nonce N'_b. In message 5, A returns this nonce encrypted under K_{ab}. Now both A and B should accept K_{ab} as the session key.

To implement this protocol using our API, A should initially have a handle $h^r_A(N'_{K_{as}}, K_{as}, 3, \{A, S\})$ to the key K_{as} of level 3. We will therefore assume that any initial state $\mathcal{S}$ is such that $h^r_A(N'_{K_{as}}, K_{as}, 3, \{A, S\}) \in \mathcal{S}_A$.

The agent A can execute its first protocol rule by using the following API command:

$$\stackrel{N,N_a}{\Rightarrow} P_A(N_a), P_A(h^g_A(N, N_a, 0, \mathsf{All})) \qquad \textbf{(Public Generate)}$$

where N, N_a are nonce variables. A runs this command with no input and obtains both a fresh (public) nonce N_a and a handle $h^g_A(N, N_a, 0, \mathsf{All})$ for it. N is a fresh value that allows to distinguish between several handles pointing to the same value N_a. Intuitively, it corresponds to the fact that the API has to generate a fresh new handle.

A's second step in the protocol (rule 5) can also be performed using the API's commands. Upon receiving a message of the form $\{N_a, A, K_{ab}\}_{K_{as}}, \{N_a\}_{K_{ab}}, N'_b$, A can split it into three parts x_1, x_2 and x_3. Intuitively, x_1 should correspond to $\{N_a, b, K_{ab}\}_{K_{as}}$, the part x_2 should correspond to $\{N_a\}_{K_{ab}}$ and x_3 should correspond to N'_b. Then A can decrypt x_1 using the following decryption command (with $L = \{1\}$, that is the first component should be checked):

$$\begin{array}{r} P_A(h^r_A(N'_{K_{as}}, K_{as}, 3, \{A, S\})), P_A(\{N_a, 0, y, 0, x, 2, \{A, B, S\}\}_{K_{as}}), \\ P_A(h^g_A(N, N_a, 0, \mathsf{All})) \\ \stackrel{N'}{\Rightarrow} P_A(y), P_A(h^r_A(N', x, 2, \{A, B, S\})) \end{array} \qquad \textbf{(Decrypt)}$$

where $N, N_a, N'_{kas}, K_{as}, x, y$ are variables. Recall that $m, 0$ is a notational convention for $m, 0, \mathsf{All}$. By applying this command to the entries $h^r_A(N'_{K_{as}}, K_{as}, 3, \{A, S\})$ (the handle referring to K_{as}), x_1 (the first component of the received message), and $h^g_A(N, N_a, 0, \mathsf{All})$ (the handle referring to N_a), A can check that y is equal to B and receives a handle $h^r_A(N', x, 2, \{A, B, S\})$ referring to some value x that should correspond to the inside key K_{ab}. Then A can decrypt x_2 using the following decryption command (with again $L = \{1\}$, that is the first component should be checked):

$$P_A(h^r_A(N', K_{ab}, 2, \{A, B, S\})), P_A(\{N_a, 0\}_{K_{ab}}), P_A(h^g_A(N, N_a, 0, \mathsf{All})) \Rightarrow \qquad \textbf{(Decrypt)}$$

where N, N_a, N', K_{ab} are variables. Note that this rule has no right-hand side: the agent does not get anything by applying this command, it is instead used as a validity check.

Indeed, if the command succeeds applied to the entries $h^r_A(N', x, 2, \{A, B, S\})$ (the handle that should correspond to K_{ab}), x_2 (the second component of the received message), and $h^g_A(N, N_a, 0, \mathsf{All})$ (the handle referring to N_a), then the agent A knows that the second component x_2 indeed corresponds to $\{N_a\}_{K_{ab}}$. Then A can build her message for B by using the following encryption command.

$$P_A(h^r_A(N', K_{ab}, 2, \{A, B, S\})), P_A(x_3) \Rightarrow P_A(\{x_3, 0\}_{K_{ab}}) \qquad \textbf{(Encrypt)}$$

where N', K_{ab} are variables.

The steps of principles B and S can similarly be implemented by our API.

4. Security of the API

Recall that our API is designed to be used on a device which may sometimes be connected to a corrupted host machine, and sometimes to a 'clean' machine. When all machines involved in a run of a protocol are 'clean' the formal threat model reduces to the so-called Dolev-Yao model: all network traffic goes through the attacker, but computations on honest users' machines remain secure. In this case, our API merely implements the protocol, and does not provide extra security. We are interested in what guarantees our API offers when one or more of the machines involved in a protocol run are under the control of an attacker. We will state and prove security properties for a certain group of users H, called by convention *honest agents*. These are agents whose host machine may be controlled by the attacker, but whose TRD is intact. Other users not in H are called *compromised*. The attacker not only controls the host machine but also can read the contents of the TRD of compromised users (i.e. he has defeated the tamper-resistance, see Figure 2). H is assumed to be a fixed set in the remainder of the paper.

If the attacker controls a host machine then all the public data on the machine (level 0 terms in our model) are assumed to be lost. We want to show that secret terms (level ≥ 1) on honest users' TRDs remain secret. We do this by means of three security invariants. In the first (called **Sec** below), we show that secret terms which are stored along with a set of users S that is a subset of H remain secret from the attacker. In the second we assume that an honest user may lose some, but not all, of his secret terms (perhaps by a side channel attack on specific keys, for example). In this case we show that other keys stay secure unless they are of strictly lower security level and are stored for a set of users larger than or equal to the lost keys (**SecDegree**). Finally, we assume the attacker is able to obtain some short term keys (level 2) and secret nonces (level 1), and show that provided the honest users refresh their TRDs in a way we will define, newly established session keys will be secure (**SecFresh**).

Intuitively, all these properties hold because of the tagging scheme used in the encryption and the ordering on this scheme enforced by the encryption and decryption rules. This ensures that even if a key is compromised, the damage the attacker can do is limited. He cannot, for example, re-import a compromised key via the decrypt command and give it a higher level l or a smaller set S, since to do so would require prior knowledge of another

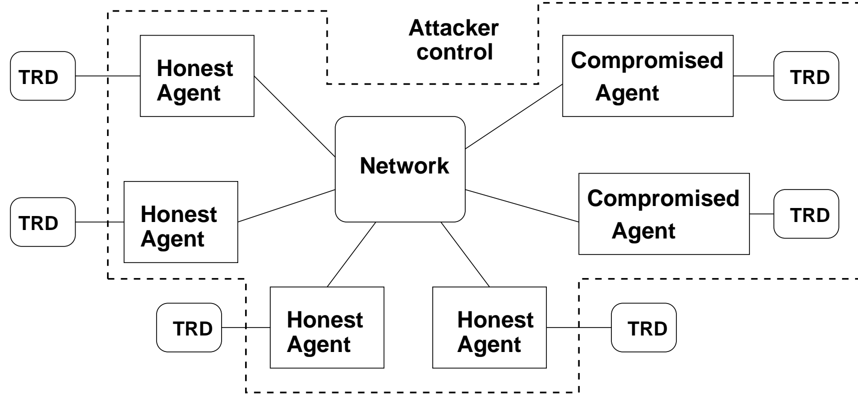

Figure 2: Threat model. The attacker controls the network, all machines, and has obtained access to the memory of some compromised agents' TRDs

key that is itself already higher in the hierarchy described by the levels l and the sets of agents S. Thus he is limited to the discovery of keys which are lower in the hierarchy than the compromised key. Note that a more refined ordering could be used to limit further the consequences of key compromise, but we leave this for future work.

We first give a precise formal model of the threat scenario. The attacker has access to the machines of all users and to the long term secrets (i.e. the contents of the TRD) of the compromised users. The only trusted secure parts are the secure storage components (TRDs) of the honest users, managed by the API. This can be easily modeled by adding the following set $\mathsf{CONTROL}$ of rules

$$\begin{array}{rcl} P_a(x) & \Rightarrow & P_{\mathsf{int}}(x) \\ P_{\mathsf{int}}(x) & \Rightarrow & P_a(x) \\ P_b(h_b^\alpha(x,y,i,S)) & \Rightarrow & P_{\mathsf{int}}(y) \end{array} \qquad (\mathsf{CONTROL})$$

for any $a, b \in \mathsf{Agent}$ such that $b \notin H$, $i \in \{1,2,3\}$, $\alpha \in \{r, g\}$ and $S \subseteq \mathsf{Agent}$. The first and second rules model the fact that even honest agents may plug their TRD in to possibly compromised machines. Thus the attacker may learn anything an agent knows (first rule), even for the honest ones, and may try to communicate with any honest API (second rule). Intuitively, agents shall not store sensitive information on their computer as sensitive information should be stored on TRD. Of course, the attacker is not given access to the content of the TRD of honest agents. In contrast, the last rule indicates the fact that the attacker is given any value that may be stored in a TRD of a compromised agent. Given a state $\mathcal{S}$ of our execution model and by abuse of notation, we write $t \in \mathcal{S}$ (resp. $\mathcal{S} \vdash t$) instead of $t \in \bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b$ (resp. $\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \vdash t$). The fact that the attacker has access to the local states of users, even honest ones, models the fact that the attacker may have corrupted honest users' machines. Of course, this does not give him access to the keys stored in the TRD.

4.1. Security for honest users

When the API is initialized, keys of level 3 are generated and distributed between the secure components managed by APIs and users are given handles to these keys. These keys are initially unknown to the attacker. Thus we say that a state $\mathcal{S}$ is *initial* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key}$ is a set of atomic messages (that is only constant terms, typically nonces, keys, or agent names) and if for any $a \in \mathsf{Agent}$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n, k, i, S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that

- n, k do not appear in $\mathcal{S}_{\mathsf{int}}$,
- $h_a^\alpha(n, k, i, S) \in \mathcal{S}_a$, $h_b^{\alpha'}(n', k, i', S') \in \mathcal{S}_b$, $a, b \in H$ imply $i = i'$ and $S = S'$: honest tokens are consistently set up.

The security of the API can be expressed as follows: given a state $\mathcal{S}$ of the system, secret data of honest users should not be known to the attacker. Secret data of honest users are values k for which there are handles of the form $h_a^\alpha(n, k, i, S)$ where S is a subset of honest users. This is reflected by the following formula:

$$\forall a \in H, \forall x, y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r, g\}, \forall S \subseteq H \quad \mathcal{S} \vdash h_a^\alpha(x, y, i, S) \Rightarrow \mathcal{S} \not\vdash y \quad (\textbf{Sec})$$

The formula **Sec** states that whenever the system contains a handle $h_a^\alpha(x, y, i, S)$ for a value y and a set S of honest agents, then y is guaranteed to be secret.

We can show that our generic API satisfies the security property **Sec** as the API is correctly initialized. This is an important feature since it guarantees confidentiality of sensitive data for our API even if the attacker has control of all honest users' machines.

Theorem 1. *Let $\mathcal{S}_0$ be an initial state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof: (sketch) We first start by adding more power to the attacker, providing him an immediate access to any value m stored on a compromised device and providing him access to any value m for which there exists a handle $h_a^\alpha(n, m, i, S)$ where some participant of S is compromised, even if a is honest, meaning that the value m is stored on a non-compromised TRD. Formally, we write $\mathcal{S} \vdash^* t$ if

$$\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \cup \{m \mid h_a^\alpha(n, m, i, S) \in \mathcal{S}, S \not\subseteq H, a \in \mathsf{Agent}\}$$
$$\cup \{m \mid h_a^\alpha(n, m, i, S) \in \mathcal{S}, a \notin H\} \vdash t \quad (\text{Definition of } \vdash^*)$$

We then consider a stronger version of property **Sec**.

$$\forall a \in H, \forall x, y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r, g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash^* h_a^\alpha(x, y, i, S) \Rightarrow \mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\textbf{Sec*})$$

Intuitively, the property **Sec*** ensures in addition to **Sec** that values stored for honest agents are always of type Nonce or Key, and are non-deducible, even for the stronger version of deducibility $\vdash^*$.

We show in Appendix A that **Sec*** is invariant under applications of the rules of $\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}$. Our proof requires proving that the three following properties are also invariant:

$$\begin{aligned}
&\forall u, k \in \mathsf{Msg}, \mathcal{S} \vdash^* \{u\}_k \Rightarrow \mathcal{S} \vdash^* k \text{ or} \\
&\quad \exists n \in \mathsf{Msg}, \exists S \subseteq H, \exists i \in \{0,1,2,3\}, \exists a \in H, \exists \alpha \in \{r,g\}, \text{ s.t. } \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \\
&\quad\quad \exists m_1, \ldots, m_p \in \mathsf{Msg}, \exists i_1, \ldots, i_p \in \{0,1,2,3\}, \\
&\quad\quad \exists S_1 \ldots, S_p, \text{ s.t. } u = i_1, S_1, m_1, \ldots, i_p, S_p, m_p, \text{ and } \forall j, S \subseteq S_j, \text{ and} \\
&\quad\quad\quad \forall i_j \geq 1, m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and} \\
&\quad\quad\quad \exists n_j \in \mathsf{Nonce}, \exists b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^* h_b^{\alpha'}(n_j, m_j, i_j, S_j) \quad (\textbf{Enc})
\end{aligned}$$

Intuitively, property **Enc** ensures that any encrypted message $\{u\}_k$ sent over the network is either built by the attacker (i.e. the attacker knows k), or corresponds to the output of an API, with the corresponding handles for each value.

$$\begin{aligned}
&\forall k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i_1, \ldots, i_p \in \{0,1,2,3\}, \forall j \\
&\quad [(i_j = 0 \text{ or } S_j \not\subseteq H) \text{ and } \mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k] \Rightarrow \mathcal{S} \vdash^* m_j \quad (\textbf{Enc0})
\end{aligned}$$

Intuitively, property **Enc0** ensures that for any encrypted message, any underlying plaintext value that is indicated as security level 0 or accessible to a dishonest agent is indeed a public value.

$$\begin{aligned}
&\forall n, n', m \in \mathsf{Msg}, \forall i, i' \in \{0,1,2,3\}, \forall \alpha, \alpha' \in \{r,g\}, \forall a, b \in H \\
&\quad \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \text{ and } \mathcal{S} \vdash^* h_b^{\alpha'}(n',k,i',S') \Rightarrow i = i' \text{ and } S = S' \quad (\textbf{SecDegree})
\end{aligned}$$

Intuitively, property **SecDegree** ensures that honest tokens agree on security levels and sets of agents being granted access to common values. This last property is not needed for proving **Sec** but will be needed in the next section.

Theorem 1 then easily follows since any initial state satisfies the four properties **Sec***, **Enc**, **Enc0**, and **SecDegree** and property **Sec** is an immediate consequence of property **Sec***.

4.2. Security of the API under compromised handles

We have seen in the previous section that our API protects any data for which there is an honest handle $h_a^\alpha(n,k,i,S)$ with $S \subseteq H$. Imagine that some secret data is leaked to the attacker by some out of model means, possibly using a brute force attack, a physical side-channel attack or some other means. So, the attacker knows both $h_a^\alpha(n,k,i,S)$ and

k. Then the attacker can learn any data of security level strictly smaller than the security level i of k, stored by the API of a, for which he has a handle $h_a^{\alpha'}(n', k', j, S')$ with $j < i$, $S \subseteq S'$. Indeed, the attacker can use the encryption command of the API

$$\mathsf{Encrypt} \quad h_a^{\alpha}(n, k, i, S) \quad h_a^{\alpha'}(n', k', j, S')$$

and obtain the cyphertext $\{j, S', k'\}_k$ thus k'. Note that this attack requires the attacker to control the API of a and only allows handles of strictly lower security level to be compromised, for values that are granted to sets of participants that include all agents having a granted access to the lost key k.

We show in this section that, even when some keys are lost, all other keys are still protected, except those that are of strictly lower level for larger sets of participants.

Let K be a set of keys (or nonces) that represent the values learned by the attacker (e.g. by brute force attacks) and let $\mathcal{S}$ be a state of the system. We define the security degree of K in $\mathcal{S}$ to be the set $\mathsf{SecDegree}(K, \mathcal{S})$ of security degrees of keys in K as they appear in handles of honest tokens.

$$\mathsf{SecDegree}(K, \mathcal{S}) = \{(i, S) \mid \exists n \in \mathsf{Msg}, \exists a \in H, \exists k \in K, \exists \alpha \in \{r, g\} \quad h_a^{\alpha}(n, k, i, S) \in \mathcal{S}\}$$

Note that if a key k appears with a certain level (i, S) in a state S then when the system evolves, the key k still appears with level (i, S) since keys are never removed from a TRD. We say that a security degree (i_1, S_1) is strictly smaller than (i_2, S_2), denoted by $(i_1, S_1) < (i_2, S_2)$, if $i_1 < i_2$ and $S_2 \subseteq S_1$. By abuse of notation, we write $(i_1, S_1) < \mathsf{SecDegree}(K, \mathcal{S})$ if there exists $(i_2, S_2) \in \mathsf{SecDegree}(K, \mathcal{S})$ such that $(i_1, S_1) < (i_2, S_2)$. We have seen that knowing the keys of K, the attacker can easily learn any key of strictly smaller degree than keys in K. We can show that the confidentiality of any other key is still preserved by our generic API.

Theorem 2. *Let $\mathcal{S}_0$ be an initial state. Let $\mathcal{S}_1$ be any state accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}_1$. Let K be any set of keys and nonces and let $\mathcal{S}'_1$ be the state $\mathcal{S}_1$ in which the intruder learns all keys in K, that is $\mathcal{S}'_{1,a} = \mathcal{S}_{1,a}$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}'_{1,\mathsf{int}} = \mathcal{S}_{1,\mathsf{int}} \cup K$. Then, for any state $\mathcal{S}$ accessible from $\mathcal{S}'_1$, that is $\mathcal{S}'_1 \Rightarrow^*_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that the following property* **SecW** *is satisfied by $\mathcal{S}$:*

$$\forall a \in H, \forall x, y \in \mathsf{Msg}, \forall i \in \{1, 2, 3\}, \forall \alpha \in \{r, g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash h_a^{\alpha}(x, y, i, S) \text{ and } (i, S) \not< \mathsf{SecDegree}(K, \mathcal{S}_1) \Rightarrow \mathcal{S} \not\vdash y \text{ or } y \in K \quad (\mathbf{SecW})$$

Theorem 1 states that, even if some keys are lost (here the set of keys K), then keys that are not strictly smaller than a lost key remain protected, that is, they cannot be learned by an attacker. In other words, the key hierarchy ensures a certain independence between keys.

Proof: (sketch) As for the proof of Theorem 1, we first give more power to the intruder, providing him access to any value m stored on a dishonest device and to any value m for which there exists a handle $h_a^\alpha(n,m,i,S)$ where some participant of S is dishonest or where $(i,S) < \mathsf{SecDegree}(K,\mathcal{S}_1)$, even if a is honest, meaning that the value m is stored on a non-compromised TRD.

Formally, we write $\mathcal{S} \vdash^W t$ if

$$\bigcup_{b\in\mathsf{Agent}\cup\{\mathsf{int}\}} \mathcal{S}_b \cup \{m \mid h_a^\alpha(n,m,i,S) \in \mathcal{S}, S \not\subseteq H, a \in \mathsf{Agent}\} \\ \cup \{m \mid h_a^\alpha(n,m,i,S) \in \mathcal{S}, a \notin H\} \\ \cup \{m \mid h_a^\alpha(n,m,i,S) \in \mathcal{S}, (i,S) < \mathsf{SecDegree}(K,\mathcal{S}_1)\} \quad \vdash t \quad \text{(Def: } \vdash^W)$$

We prove that a stronger invariant is preserved by our API. We consider the three following properties:

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H \\ \mathcal{S} \vdash^W h_a^\alpha(x,y,i,S) \text{ and } (i,S) \not< \mathsf{SecDegree}(K,\mathcal{S}_1) \Rightarrow y \in K \text{ or} \\ \mathcal{S} \not\vdash^W y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\mathbf{SecW*})$$

$$\forall u,k \in \mathsf{Msg}, \mathcal{S} \vdash^W \{u\}_k \Rightarrow \mathcal{S} \vdash^W k \text{ or} \\ \exists n \in \mathsf{Msg}, \exists S \subseteq H, \exists i \in \{0,1,2,3\}, \exists a \in H, \exists \alpha \in \{r,g\}, \text{ s.t. } \mathcal{S} \vdash^W h_a^\alpha(n,k,i,S) \\ \exists m_1,\ldots,m_p \in \mathsf{Msg}, \exists i_1,\ldots,i_p \in \{0,1,2,3\}, \\ \exists S_1.\ldots,S_p, \text{ s.t. } u = i_1,S_1,m_1,\ldots,i_p,S_p,m_p, \text{ and } \forall j, S \subseteq S_j, \text{ and} \\ \forall i_j \geq 1, m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and} \\ \exists n_j \in \mathsf{Nonce}, \exists b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^W h_b^{\alpha'}(n_j,m_j,i_j,S_j) \quad (\mathbf{EncW})$$

$$\forall k,m_1,\ldots,m_p \in \mathsf{Msg}, \forall i_1,\ldots,i_p \in \{0,1,2,3\}, \forall j \text{ s.t. } i_j = 0 \text{ or } S_j \not\subseteq H \\ \mathcal{S} \vdash^W \{i_1,S_1,m_1,\ldots,i_p,S_p,m_p\}_k \Rightarrow \mathcal{S} \vdash^W m_j \quad (\mathbf{Enc0W})$$

We show in Appendix B (by a careful inspection of the rules) that properties **SecW***, **EncW** and **Enc0W** are invariant under application of the rules of $\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}$. We also show in Appendix B that $\mathcal{S}'_1$ satisfies the three properties **SecW***, **EncW** and **Enc0W**.

Theorem 2 then easily follows since property **SecW** is an immediate consequence of property **Sec***.

4.3. Security of the API with periodic erasure

We have seen in the previous section that our API still preserves some security when keys are lost. This situation is not completely satisfactory since some groups of agents are definitively not protected.

Thus we assume that (honest) agents periodically erase from the API any handle that corresponds to a datum of a security level strictly lower than 3. Since data of security level 2 are typically session keys and data of security level 1 are typically nonces, it makes sense to refresh them periodically. Formally, we say that a state $\mathcal{S}$ is *refreshed* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Msg}$ is any set of messages and if for any $a \in H$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n, k, 3, S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that k only (possibly) appears in $\mathcal{S}$ in key position[1] whenever $S \subseteq H$. Note that we do not make any assumption on the states of compromised agents (besides that honest keys of level 3 only appear in key position).

This is however still not sufficient to guarantee the security of the API in case the attacker is able to learn old keys. Indeed, assume that an attacker knows a cyphertext $\{j, S', k'\}_k$ where k is a long-term (honest) key (of security level 3) such that he also knows k' (possibly using brute force attacks) of security level 2. For every (honest) agent a that has access to k using some handle of the form $h_a^r(n, k, 3, S)$, the attacker can register k' using the decryption command of the API of a.

$$\mathsf{Decrypt} \quad h_a^r(n, k, 3, S) \quad \{j, S', k'\}_k$$

The attacker then learns $h_a^\alpha(n', k', 2, S')$, a fresh handle that refers to k', which allows him to mount the previous attack, again allowing the attacker to learn any data of security level 1 stored by the TRD of a. This corresponds to a classical replay attack. Intuitively, since our API can be used to implement a protocol subject to replay, it suffers from replay attack as well.

To prevent such replay attacks, we reinforce the security of the API by restricting the use of decryption rules: the API should allow decryption with keys of level 3 only if at least one component is checked for freshness. In particular, our restricted API will not allow the implementation of protocols subject to this form of replay attack. Formally this corresponds to considering only decryption rules of the form

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in L} P_a(m'_j) \qquad \stackrel{N_1, \ldots, N_p}{\Rightarrow} \qquad \bigcup_{j \notin L} P_a(m'_j)$$

where L must not be the empty set whenever $i_0 = 3$ (and all the other conditions of the decryption rule of Figure 1 are fulfilled). Let API^r be the set of rules obtained from API by removing the decryption rules where L is empty when $i_0 = 3$.

Our restricted API preserves secrecy of its confidential values, even when the attacker

[1] That is, whenever k occurs at position p in a message t of $\mathcal{S}$, then $p = p'.2$ and $t|_{p'} = \{t'\}_k$.

is able to learn old keys and to control honest APIs, provided honest agents have refreshed the data in their TRDs.

Theorem 3. *Let $\mathcal{S}_0$ be a refreshed state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API}^r \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof (sketch): Let $\mathcal{S}_0$ be a refreshed state. We define Fresh to be the set of *fresh* values, that is the set of nonces and keys that do not occur in $\mathcal{S}_0$. As for the proof of Theorem 1, we first re-enforce the properties that are invariant under $\mathsf{API}^r \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}$. We consider the three following properties.

$$\forall a \in H, \forall x, y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall S \subseteq H, \forall \alpha \in \{r,g\},\ \mathcal{S} \vdash^* h^\alpha_a(x,y,i,S) \ \Rightarrow \\ \mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and in case } i \neq 3 \text{ then } y \in \mathsf{Fresh} \quad (\textbf{SecFresh*})$$

$$\forall n, k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i, i_1, \ldots, i_p \in \{0,1,2,3\}, \forall \alpha \in \{r,g\}, \forall a \in H, \forall j \\ i_j \geq 1, S_j \subseteq H, \forall S \subseteq H, \mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k, \mathcal{S} \vdash^* h^\alpha_a(n,k,i,S) \Rightarrow \\ (m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and } \exists n_j \in \mathsf{Nonce}, b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^* h^{\alpha'}_b(n_j, m_j, i_j, S_j)) \\ \text{or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad (\textbf{Enc'})$$

$$\forall k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i_1, \ldots, i_p \in \{0,1,2,3\}, \forall j \text{ s.t. } i_j = 0 \text{ or } S_j \not\subseteq H \\ \mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \Rightarrow \\ \mathcal{S} \vdash^* m_j \text{ or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad (\textbf{Enc0'})$$

We show in Appendix C (by a careful inspection of the rules) that these three properties are invariant under application of the rules of $\mathsf{API}^r \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}$. Theorem 3 then easily follows since any refreshed state satisfies the three properties **SecFresh***, **Enc'** and **Enc0'** and property **Sec** is an immediate consequence of property **SecFresh***.

Note that our freshness condition does not require agents to erase their data after each session. Intuitively, refreshment should occur only when a leak from an honest user is suspected or when keys have been stored and used for a sufficient time to allowing brute force attacks. Thus refreshment could occur every hour, day, week, month or year depending on application-specific factors.

5. Using the Generic API to Implement a Protocol

In this section we show how our generic API can be used to implement symmetric key protocols, including in particular symmetric key distribution protocols from the venerable Clark-Jacob survey [7].

To deduce the API commands, we first require the protocol to be specified in a manner following e.g. [21], that is each protocol step is given as a rule

$$A : u \xrightarrow{new\ \mathcal{N}} v$$

A is the agent who plays the role. The u, v are terms in our algebra from section 3, where agent names, keys and nonces are given as variables. The set $\mathcal{N}$ of nonce and key variables represents freshly generated data. In addition we require the terms in the protocol to be tagged with their type (agent, nonce, key or message), and nonces and session keys must be tagged with the name of the agent which generated them, their level (0 for a nonce sent in the clear, 1 for a nonce only ever sent encrypted, 2 for a session key) and the set of participants expected to share secrets. Everything generated by the participants during the protocol (i.e. keys and nonces) will be assumed to be shared between all participants. We will not attempt to deduce whether a nonce is kept secret from the server, or secret from Bob, etc. Tagged nonces in a protocol will be written $n(A, N_A, L, Set)$, where A is the agent, N_A the name for the nonce, L the level and Set the set. Similarly, we have tagged keys $k(S, K_A, L, Set)$, agent names $a(A)$ and message variables $m(X)$. This tagging can be easily guessed by a user reading the protocol but could also be found automatically (for example, by trying several possible taggings).

Given a tagged term t, $\mathsf{un}(t)$ denotes its untagged version obtained from t by removing all the tags. For example, $\mathsf{un}(n(A, N_A, L, Set)) = N_A$. Moreover, given a term t, we denote by $\bar{t}$ the term obtained from t by replacing each subterm $\{u\}_v$ of t by the variable $X_{\{u\}_v}$. The function $\bar{\cdot}$ is a one-to-one mapping.

5.1. Algorithm

We give a simple algorithm for constructing API commands for a given protocol below in informal pseudocode. The algorithm relies on a global store H of handles that each participant in the protocol will expect to have when a protocol step is executed. This store has an initial state. For example, for the three-party key exchange protocols, the initial state is

$h^r_a(N_{Kas}, kas, 3, \{a, s\})$ `% A handle for kas`
$h^r_b(N_{Kbs}, kbs, 3, \{b, s\})$ `% B handle for kbs`
$h^g_s(N'_{Kas}, kas, 3, \{a, s\})$ `% S handle for kas`
$h^g_s(N'_{Kbs}, kbs, 3, \{b, s\})$ `% S handle for kbs`

Note that where we give agent names a, b, and s as ground terms these should be interpreted as parameters - it is up to the implementer to equip the TRD with the handles and API for the roles of a, b or s as appropriate. This "personalisation phase", where the long term keys are loaded onto the smartcard in a secure environment, is common practice in the use of such devices.

To implement the protocol, for each rule $u \xrightarrow{new\,\mathcal{N}} v$ played by agent A, we synthesize in turn:

1. zero or more Decryption Commands, followed by
2. zero or more Generate commands, followed by
3. zero or more Encryption Commands

Decryption

For each encyption $\{m_1, \ldots, m_p\}_{Xk}$ occurring in u:

1. Retrieve $h^{\alpha}_{A}(N, X_k, j, Set)$ from store H. If none exists then the algorithm fails. The protocol is actually not executable since the agent does not have the decryption key (and encrypted packets for forwarding must be marked as message variables).
2. Set $L = []$. For all m_i such that $m_i = n(A, X, I, Set)$ and $h^{g}_{A}(N', X, I, Set)$ is in the handle store and set $L := L \cup \{i\}$. If no such m_i exists, and $j = 3$ then output the warning "missing freshness test". We have seen that tests ensure a higher level of security.
3. Add a decryption command of the form

$$P_A(h^{\alpha}_{A}(N, X_k, j, Set)), P_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_p)}\}_{X_k}), L \overset{N_1,\ldots,N_p}{\Rightarrow} \bigcup_{j \neq i} P_A(m'_i)$$

where the m'_i are defined from the $\overline{\mathsf{un}(m_i)}$ as in section 3.3.

Generate

For each $n(A, X, 0, Set) \in \mathcal{N}$:

1. Add a generate command

$$\overset{N,X}{\Rightarrow} P_A(X), P_A(h^{g}_{A}(N, X, L, Set))$$

2. Add $h^{g}_{A}(N, X, 0, Set)$ to the handle store H.

For each $n(A, X, i, Set) \in \mathcal{N}$, $i \in \{1, 2\}$:

1. Add a generate command

$$\overset{N,X}{\Rightarrow} P_A(h^{g}_{A}(N, X, i, Set))$$

2. Add $h^{g}_{A}(N, X, i, Set)$ to the handle store H.

Encryption

For each encryption $\{m_1, \ldots, m_p\}_{Xk}$ occurring in v:

1. Retrieve $h^{\alpha}_{A}(N, X_k, i, Set)$ from the handle store H. If no such handle exists, fail.
2. Add an encryption command of the form

$$P_A(h^{\alpha}_{A}(N, X_k, i, S)), P_A(m'_1), \ldots, P_A(m'_k) \Rightarrow P_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_k)}\}_{X_k})$$

   where m'_i is

   - h if $m_i = n(A, Y, 1, S)$ is a level 1 nonce with a handle $h = h^{\alpha}_{A}(N', Y, 1, S) \in H$
   - h if $m_i = k(A, X, 2, S)$ is a key with a handle $h = h^{\alpha}_{A}(n', Y, 2, S) \in H$
   - $\overline{\mathsf{un}(m_i)}$ if m_i is an agent name, a nonce of level 0, a message variable or a cyphertext.
   - The algorithm fails otherwise, that is, in case m_i is of level security 1 or 2 with no corresponding handle in the store (or if m_i is of higher security level). This corresponds to a case where the agents is enable to build the message thus the protocol is not executable.

   We consider encrypted terms to be terms of level 0. In this way we can treat nested encryptions by recursively generating encryption commands, treating the innermost encryption first.

5.2. Example

Consider the role of A in the Carlsen's Secret Key Initiator Protocol [5, Figure 2]. Using our algorithm, we retrieve the API commands presented in example 3.4.

5.3. Results

A Prolog implementation has been tested on all the protocols in section 6.3 of the Clark-Jacob survey, excepting those where freshness is assured by timestamps. The Prolog source and the results are given in Appendix D and available via http [2]. We summarise the results in Table 1. The results illustrate how the properties we are able to guarantee by the use of our API translate to the properties of the protocols that can be implemented. Needham-Schroeder Symmetric Key can be implemented by API but not $\mathsf{API}^{\mathsf{r}}$ (the restricted mode of our API, described in Section 4.3), and indeed is subject to a replay attack. The amended version can be implemented by $\mathsf{API}^{\mathsf{r}}$, and has no known attack. The Otway-Rees protocol has a known type attack, which would be avoided by the tagged encryption scheme used by our API since in particular agent identities are included in every encryption. The Yahalom protocol cannot be implemented by $\mathsf{API}^{\mathsf{r}}$. The missing test is reported for the final message to B. At first sight this would seem to indicate inadequate functionality in our API, since

[2] `http://www.lsv.ens-cachan.fr/~steel/GenericAPI/`

B is supposedly assured of the freshness of the session key by the fact that A has used it to encrypt B's nonce in a separate packet. However, this missing test can in fact be exploited by a malicious party playing A's role in the protocol to force B to accept an old key [19]. Carlsen's protocol has no known attack. Woo-Lam has a known parallel session attack, but this exploits a type flaw which our encryption scheme would prevent.

Note that there are secure protocols that could not be implemented by our API in its current form. These include for example protocols which use a nonce as a secret value in an initial exchange but then reveal it later (our API requires the security level of data to be fixed), or protocols that use a more subtle notion of freshness. However, the explicitness forced by our API is a well-known good design principle for protocols [1].

Protocol (section in Clark-Jacob)	API	API^r
Needham-Schroeder SK (6.3.1)	+	-
NSSK amended version (6.3.4)	+	+
Otway-Rees (6.3.3)	+	+
Yahalom (6.3.6)	+	-
Carlsen (6.3.7)	+	+
Woo-Lam Mutual Auth (6.3.11)	+	+

Table 1: Implementation of some protocols. API is the original API (see section 2), and API^r is the restricted API where we insist on at least one test for every new session key (see section 4.3). A + indicates an implementation of the protocol was found by our algorithm in section 5. A - indicates the algorithm reported a missing test.

6. Conclusions

We have presented a generic API for key management on tamper-resistant devices. We have shown how it can be used to implement a variety of symmetric key protocols. We have given a formal security policy that the API will preserve under a variety of given threat scenarios, no matter what calls the attacker makes to the API. In particular, in a scenario where an attacker may learn old secret values such as nonces and session keys, our API can be run in a restricted mode where checks are made for freshness before accepting a new key. In this mode, fewer protocols can be implemented, but we exclude protocols susceptible to replay attacks.

Although our API is limited to symmetric key cryptography and a particular notion of freshness checking which may not accommodate all correct protocols, we believe we have established that it is possible to construct a secure API with a satisfactory level of generality by examining the protocols it is supposed to implement. Extensions to asymmetric cryptography, time-based notions of freshness and explicit revocation are all ongoing work. In particular, this last extension will allow us to relax our assumptions on simultaneous 'refreshing' of devices to regain security after some keys are lost. Note also that all our proofs are in the so-called 'symbolic model', where encryption is treated as a black box

function on terms. We intend to investigate the extension of our results to more precise computational models of security.

As we mentioned in the introduction, most previous work on analysis of security APIs has resulted in the discovery of flaws in existing schemes. Some positive results include the verification of various fixes of the IBM CCA for a particular security property (the secrecy of PINs) [9, 11, 22, 16]. Note that all of these results, including the work by Courant and Monin in Coq, are for a bounded model, where the attacker has just one session key (called a data key in the CCA) and one long-term key (Key-encrypting key). Work by the second author includes the verification of the secrecy of sensitive keys for a small subset of PKCS#11 in an unbounded model [13]. The resulting API is for one user only and provides no security if a single long-term key is lost. Ours is the first API to propose a general scheme for bringing nonce-based freshness checks within the 'trust boundary' of the tamper-resistant device. Other than our own design, the most sophisticated API with a security proof is the Cachin-Chandran API [4], which as we saw in section 2, provides a solution for a different scenario to our API: security is based on a history of operations, and so requires a single central log. Furthermore, the API requires that there is only one single copy of each key in existence, and all keys are known to the single server, thus avoiding problems of key freshness. While this is suitable for a central key management server, where the keys may be used e.g. to encrypt data for storage or to sign documents, our API seems more suitable for distributed tokens, where there may be several copies of each key on several different devices, and so the security policy for each key has to be fixed. Indeed this difference between fixing the policy and allowing the policy to evolve over time seems to be a rather fundamental distinction based on the scenario in which the API is to be applied. An open challenge is to combine these designs to accommodate a system with one or more central key servers and a number of distributed tokens, where the servers offer more functionality than the distributed devices based on their greater storage resources. Such a design would fit more closely the architecture of real-world embedded systems that either of the two APIs alone. We plan to tackle this in future work.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement number 258865 (ERC ProSecure project) and from the Direction Générale de l'Armement. Work partially carried out while the second author was at LSV, CNRS & INRIA & ENS de Cachan.

References

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1), January 1996.

[2] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[3] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269, Chicago, Illinois, USA, Oct. 2010. ACM Press.

[4] C. Cachin and N. Chandran. A secure cryptographic token interface. In *Computer Security Foundations (CSF-22)*, pages 141–153, Long Island, New York, 2009. IEEE Computer Society Press.

[5] U. Carlsen. Optimal privacy and authentication on a portable communications system. *SIGOPS Oper. Syst. Rev.*, 28(3):16–23, 1994.

[6] *CCA Basic Services Reference and Guide*, Oct. 2006. Available online at `www.ibm.com/security/cryptocards/pdfs/bs327.pdf`.

[7] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[8] J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Worshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425, Cologne, Germany, 2003. Springer.

[9] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 538–552, Braga, Portugal, 2007. Springer.

[10] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In M. Backes and P. Ning, editors, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *Lecture Notes in Computer Science*, pages 605–620, Saint Malo, France, Sept. 2009. Springer.

[11] J. Courant and J.-F. Monin. Defending the bank with a proof assistant. In *Proceedings of the 6th International Workshop on Issues in the Theory of Security (WITS'06)*, pages 87 – 98, Vienna, Austria, March 2006.

[12] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[13] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Proceedings of ARSPA-WITS '09*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2009.

[14] IBM Comment on "A Chosen Key Difference Attack on Control Vectors", Jan. 2001. Available from `http://www.cl.cam.ac.uk/~mkb23/research.html`, 7 pages.

[15] Information Technology Laboratory. Federal information processing standards publication 140-2: Security requirements for cryptographic modules. Technical report, National Institute of Standards and Technology (NIST), 2001. Available from `csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`, 69 pages.

[16] R. Küsters and T. Truderung. Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 129–138. ACM Press, 2008.

[17] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[18] B. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[19] A. Perrig and D. Song. Looking for diamonds in the desert. In *Proc. of the 13th Computer Security Foundations Workshop (CSFW'00)*, pages 64–76. IEEE Computer Society Press, 2000.

[20] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

[21] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc. of the 14th Computer Security Foundations Workshop (CSFW'01)*, pages 174–190. IEEE Computer Society Press, 2001.

[22] R. Zunino. Defending the bank with a static analysis. In *Nordsec*, 2006. Available from `http://www.di.unipi.it/~zunino/papers/`, 12 pages.

Appendix A. Proof of Theorem 1

We first note that the intruder's power does not really increase when he learns handles or an indecipherable cyphertext.

Lemma 1. *Let $\mathcal{S}$ and $\mathcal{S}'$ be set of terms such that $\mathcal{S}' = \mathcal{S} \cup \{\{m\}_k\} \cup \mathsf{Hdls}$ where k is not deducible (i.e. $\mathcal{S} \not\vdash^* k$) and $\mathsf{Hdls} \subseteq \mathsf{Handle}$ is a set of handles such that either handles are associated to honest agents or point to an already known value. Formally, we assume*

$$\mathsf{Hdls} \subseteq \{h(n,u,i,S) \mid S \subseteq H\} \cup \{h(n,u,i,S) \mid S \not\subseteq H \text{ and } \mathcal{S} \vdash^* u\}.$$

Let $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ *be an atomic value. Then*

$$\mathcal{S}' \vdash^* u \quad \textit{implies} \quad \mathcal{S} \vdash^* u \textit{ or } u \in \mathsf{Hdls}. \tag{A.1}$$

Moreover, Let $v \in \mathsf{Msg}$ *and* $w \in \mathsf{Key}$*. Then*

$$\mathcal{S}' \vdash^* \{v\}_w \textit{ and } \mathcal{S}' \not\vdash^* w \quad \textit{iff} \quad \mathcal{S} \vdash^* \{v\}_w \textit{ or } \{v\}_w = \{m\}_k. \tag{A.2}$$

The proof mainly relies on the fact that keys are atomic.

We are now ready to show that properties **Sec***, **Enc**, **SecDegree**, and **Enc0** are invariant under application of the rules of the API. Let $\mathcal{S}$ be a state satisfying **Sec***, **Enc**, **SecDegree**, and **Enc0** and consider a state $\mathcal{S}'$ such that $\mathcal{S} \rightarrow_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$. Let us show that $\mathcal{S}'$ satisfies **Sec***, **Enc**, **SecDegree**, and **Enc0**

Let us first notice that properties **Sec***, **Enc**, **SecDegree**, and **Enc0** are clearly invariant under application of the rules of ATTACKER $\cup$ CONTROL. Indeed, if $\mathcal{S} \rightarrow_{\mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$ then for any term u, we have $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$. For the last rule of CONTROL, this is due to the fact that whenever $h_b^\alpha(n, k, i, S) \in \mathcal{S}$ with $b \notin H$ then $\mathcal{S} \vdash^* h_b^\alpha(n, k, i, S)$.

Thus we now assume $\mathcal{S} \rightarrow_{\mathsf{API}} \mathcal{S}'$ and we consider three cases depending on the rule that has been applied.

Secure and Public Generation rule. $a \in S \subseteq \mathsf{Agent}$

$$\overset{N,K}{\Rightarrow} \;\; P_a(h_a^g(N, K, i, S)) \text{ and } i \in \{1, 2\}$$
$$\overset{N,K}{\Rightarrow} \;\; P_a(K), P_a(h_a^g(N, K, 0, \mathsf{All}))$$

$\mathcal{S}' = \mathcal{S} \cup \{h_a^g(n, k, i, S)\}$ and $i \in \{1, 2\}$ or $\mathcal{S}' = \mathcal{S} \cup \{k, h_a^g(n, k, 0, \mathsf{All})\}$, where n is a fresh nonce and k is a fresh key. Using the fact that only fresh values are added, we easily check that $\mathcal{S}$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0** implies $\mathcal{S}'$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0**.

Encryption rule.

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(m_1), \ldots, P_a(m_p) \Rightarrow P_a(\{m'_1, \ldots, m'_p\}_{X_k})$$

We have $m_j\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$ and $\mathcal{S}' = \mathcal{S} \cup \{(\{m'_1, \ldots, m'_p\}_{X_k})\theta\}$ where the m_j and $m'_j = m''_j, i_j, S_j$ are defined as in rule **Encrypt**. Let $m'_j = m''_j, i_j, S_j$. If $a \notin H$ then $\mathcal{S} \vdash^* m'_i\theta$ and $\mathcal{S} \vdash^* X_k\theta$. Thus $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term or handle u. We deduce that $\mathcal{S}$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0** implies $\mathcal{S}'$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0** in the case $a \notin H$.

We now assume $a \in H$. We distinguish between two possible cases:

- Case 1: $\mathcal{S} \vdash^* X_k\theta$. In that case, since $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$, property **Sec*** ensures that $S_0 \not\subseteq H$. We deduce that for any $1 \leq i_j \leq 3$, we have $S_j \not\subseteq H$ thus $m_j\theta = h_a^\alpha(n_j, k_j, i_j, S_j)$ and $\mathcal{S} \vdash^* k_j$. For any $i_j = 0$, we have $m_j\theta \in \mathcal{S}$. Thus in both cases, we deduce that $\mathcal{S} \vdash^* m'_j\theta$ for any $1 \leq j \leq p$. This implies that

$\mathcal{S} \vdash^* (\{m'_1, \ldots, m'_p\}_{X_k})\theta$ thus the set of terms deducible from $\mathcal{S}$ is identical to the set of terms deducible from $\mathcal{S}'$. We can therefore deduce that $\mathcal{S}$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0** implies $\mathcal{S}'$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0**.

- Case 2: $\mathcal{S} \not\vdash^* X_k\theta$. Thus it must be the case that $S_0 \subseteq H$ (otherwise $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$ implies $\mathcal{S} \vdash^* X_k\theta$). Moreover, applying Lemma 1 (Equation A.1), we get that $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ (*).

  Property **Sec***: Assume $\mathcal{S}' \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $a \in H$, $E_1 \subseteq H$. Then (*) implies $\mathcal{S} \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$. Thus Property **Sec*** implies $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$. Thus applying (*) again, we get $\mathcal{S}' \not\vdash^* k_1$. We conclude that $\mathcal{S}'$ satisfies Property **Sec***.

  Property **Enc**: Assume $\mathcal{S}' \vdash^* \{u\}_k$ for some $u, k \in \mathsf{Msg}$ and assume $\mathcal{S}' \not\vdash^* k$. If $\mathcal{S} \vdash^* \{u\}_k$, we conclude using the fact that $\mathcal{S}$ satisfies Property **Enc**. Otherwise, we must have $\{u\}_k = \{m'_1, \ldots, m'_p\}_{X_k}\theta$. Thus $\mathcal{S}'$ satisfies Property **Enc**.

  Property **Enc0**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$ or $S_j \not\subseteq H$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0**. If $\mathcal{S}' \vdash^* k$ then we of course have $\mathcal{S}' \vdash^* u_j$. Otherwise, by Lemma 1 (Equation A.2), we must have $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k = (\{m'_1, \ldots, m'_p\}_{X_k})\theta$. Thus $S_j = S'_j$ and $m_j\theta = u_j \in \mathcal{S}$ if $i_j = 0$ or $m_j\theta = h(n_j, u_j, i_j, S_j)$ with $S_j \not\subseteq H$. In both cases, we deduce $\mathcal{S}' \vdash^* u_j$ and $\mathcal{S}'$ satisfies Property **Enc0**.

  Property **SecDegree** clearly holds since no handle is added to the state.

Decryption rule.

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in L} P_a(m'_j) \overset{N_1, \ldots, N_k}{\Rightarrow} \bigcup_{j \notin L} P_a(m'_j)$$

If $a \notin H$ then $\mathcal{S} \vdash^* m_i\theta$ and $\mathcal{S} \vdash^* X_k\theta$. Moreover, for any $m'_j\theta$ of the form $h(n_j, u_j, i_j, S_j)$ with corresponding $m_j\theta = u_j, i_j, S_j$, we have $a \in S_0 \subseteq S_j$ thus $S_j \not\subseteq H$ Property **Enc0** ensures that $\mathcal{S} \vdash^* u_j$. Thus applying Lemma 1, $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term u and for any handle u except if $u = m'_i\theta$ for some i and if u is a handle. We deduce that $\mathcal{S}$ satisfies properties **Enc**, **SecDegree**, and **Enc0** implies $\mathcal{S}'$ satisfies properties **Enc**, **SecDegree**, and **Enc0**. Moreover, $\mathcal{S} \vdash^* X_k\theta$ implies $S_0 \not\subseteq H$ since $\mathcal{S}$ satisfies **Sec***. Thus for any $m_j\theta$ of the form i_j, S_j, u_j we must have $S_j \not\subseteq H$. Thus $\mathcal{S}'$ satisfies **Sec*** even for the handles introduced by the decryption rule. We deduce $\mathcal{S}'$ satisfies properties **Sec***, **Enc**, **SecDegree**, and **Enc0**.

We now assume $a \in H$. We have $\{m_1, \ldots, m_k\}_{X_k}\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$, $m'_j\theta \in \mathcal{S}$ for any $j \in L$ and $\mathcal{S}' = \mathcal{S} \cup \{m'_j\theta \mid j \notin L\}$ where the m_i and m'_i are defined as in rule **Encrypt**. Let $m_j = m''_j, i_j, S_j$. For any j such that $i_j = 0$, Property **Enc0** ensures that $\mathcal{S} \vdash^* m'_j\theta$. For any j such that $i_j \geq 1$, then $m'_j\theta$ is a fresh handle of the form $h(n_k, u_j, i_j, S_j)$. Either $S_j \not\subseteq H$ and Property **Enc0** ensures that $\mathcal{S} \vdash^* u_j$ or $S_j \subseteq H$. Thus

we can deduce from Lemma 1 (Equation A.1) that for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$, we have $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ or $u = m'_j\theta$ for some j such that $i_j \geq 1$ (**).

Property **Sec***: Assume $\mathcal{S}' \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $a \in H$, $E_1 \subseteq H$. Then (**) implies $\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ or $h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j\theta$ for some j such that $i_j \geq 1$. In the first case ($\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$), Property **Sec*** ensures $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In the second case ($h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j\theta$) then $m_j\theta = k_1$. Thus we must have $S_0 \subseteq H$ (since $S_0 \subseteq E_1$ by construction of the decryption rule). Property **Sec*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^* X_k\theta$. Property **Enc** on $\mathcal{S}$ ensures that there exists $n' \in \mathsf{Nonce}$, $\alpha'' \in \{r, g\}$, $c \in H$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n', k_1, l_1, E_1)$. Property **Sec*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In both cases we conclude that $\mathcal{S}'$ satisfies Property **Sec***.

Property **Enc**: Assume $\mathcal{S}' \vdash^* \{u\}_k$ and $\mathcal{S}' \not\vdash^* k$ for some $u, k \in \mathsf{Msg}$. We have $\mathcal{S} \not\vdash^* k$. By Lemma 1 and (**), we must have $\mathcal{S} \vdash^* \{u\}_k$. Thus Property **Enc** on $\mathcal{S}$ implies that Property **Enc** holds for $\mathcal{S}'$.

Property **Enc0**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i'_j = 0$ or $S_j \not\subseteq H$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0**. Otherwise, by Lemma 1 (Equation A.2), we must have $\mathcal{S}' \vdash^* k$ thus $\mathcal{S}' \vdash^* u_j$.

Property **SecDegree**: Assume $\mathcal{S}' \vdash^* h_a^{\alpha}(n, k, i, S)$ and $\mathcal{S}' \vdash^* h_b^{\alpha'}(n', k, i', S')$. If $\mathcal{S} \not\vdash^* h_a^{\alpha}(n, k, i, S)$ then $h_a^{\alpha}(n, k, i, S)$ is one of the $m'_j\theta$, $j \notin L$ thus $m_j\theta = i, S, k$ and since $\mathcal{S}$ satisfies **Enc**, there exists a_1, α_1, n_1 such that $\mathcal{S} \vdash^* h_{a_1}^{\alpha_1}(n_1, k, i, S)$. Similarly, if $\mathcal{S} \not\vdash^* h_b^{\alpha'}(n', k, i', S')$ then $h_b^{\alpha'}(n', k, i', S')$ is one of the $m'_j\theta$, $j \notin L$ thus $m_j\theta = i', S', k$ and since $\mathcal{S}$ satisfies **Enc**, there exists b_1, α'_1, n'_1 such that $\mathcal{S} \vdash^* h_{b_1}^{\alpha'_1}(n'_1, k, i', S')$. In any case, since $\mathcal{S}$ satisfies **SecDegree**, we deduce that $\mathcal{S}'$ satisfies **SecDegree**.

Appendix B. Proof of Theorem 2

Let $\mathcal{S}_0$ be an initial state. Let $\mathcal{S}_1$ be any state accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}_1$. Let K be any set of keys and nonces and let $\mathcal{S}'_1$ be the state $\mathcal{S}_1$ in which the intruder learns all keys in K, that is $\mathcal{S}'_{1,a} = \mathcal{S}_{1,a}$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}'_{1,\mathsf{int}} = \mathcal{S}_{1,\mathsf{int}} \cup K$.

Let us first show that $\mathcal{S}'_1$ satisfies the three properties **SecW***, **EncW** and **Enc0W**. By Theorem 1, we know that $\mathcal{S}_1$ satisfies **Sec***, **Enc** and **Enc0**. Thus we can easily deduce that $\mathcal{S}'_1$ satisfies **EncW** and **Enc0W**. Let us show that $\mathcal{S}'_1$ also satisfies **SecW***. We show by induction on the length of the deduction proof (or more precisely, on the number of applications of a decryption rule) that for any $y \in \mathsf{Key} \cup \mathsf{Nonce}$, $\mathcal{S}'_1 \vdash^W y$ implies

$$\mathcal{S}_1 \vdash^* y \text{ or } y \in K \text{ or } \exists \mathsf{h}_a^{\alpha}(n, y, i, S), a \in H, (i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1), \mathcal{S}_1 \vdash^* \mathsf{h}_a^{\alpha}(n, y, i, S)$$

We consider a minimal deduction proof of $\mathcal{S}'_1 \vdash^W y$.

- Base case: if $y \in \bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}'_{1,b} \cup \{m \mid h_a^{\alpha}(n, m, i, S) \in \mathcal{S}'_1, S \not\subseteq H, a \in \mathsf{Agent}\} \cup \{m \mid h_a^{\alpha}(n, m, i, S) \in \mathcal{S}'_1, a \notin H\} \cup \{m \mid h_a^{\alpha}(n, m, i, S) \in \mathcal{S}'_1, (i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1)\} \vdash t$

then $\mathcal{S}_1 \vdash^* y$ or $y \in K$ or $\exists \mathsf{h}_a^\alpha(n, y, i, S), a \in H, (i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1), \mathcal{S}_1 \vdash^* \mathsf{h}_a^\alpha(n, y, i, S)$.

- Induction. Assume that $\mathcal{S}_1' \vdash^W \{u\}_k$, $\mathcal{S}_1' \vdash^W k$ and y is now deducible from u applying projections. By minimality of the proof, we must have $\mathcal{S}_1 \vdash^* \{u\}_k$. Moreover, by induction hypothesis, we have that $\mathcal{S}_1 \vdash^* k$ or $k \in K$ or $\exists \mathsf{h}_a^\alpha(n, y, i, S), (i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1), \mathcal{S}_1 \vdash^* \mathsf{h}_a^\alpha(n, y, i, S)$. If $\mathcal{S}_1 \vdash^* k$ then we deduce that $\mathcal{S}_1 \vdash^* y$. Otherwise $\mathcal{S}_1 \not\vdash^* k$ then by property **Enc** on $\mathcal{S}_1$, we know that there exists a handle of the form $h_a^\alpha(n, k, i, S)$, $a \in H$ such that $\mathcal{S}_1 \vdash^* h_a^\alpha(n, k, i, S)$. By inducton hypothesis and since $\mathcal{S}_1' \vdash^W k$ and $\mathcal{S}_1 \not\vdash^* k$, we have $k \in K$ or $\exists \mathsf{h}_a^\alpha(n, y, i, S), (i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1), \mathcal{S}_1 \vdash^* \mathsf{h}_a^\alpha(n, y, i, S)$. Thus in both cases, we have $(i, S) \in \mathsf{SecDegree}(K, \mathcal{S}_1)$ or $(i, S) < \mathsf{SecDegree}(K, \mathcal{S}_1)$. Moreover, still by property **Enc**, there exist $m_1, \ldots, m_p \in \mathsf{Msg}$, $i_1, \ldots, i_p \in \{0, 1, 2, 3\}$ $S_1 \ldots, S_p, S \subseteq S_j$ such that $u = i_1, S_1, m_1, \ldots, i_p, S_p, m_p$ and y is one of the $m)j$, say m_{j_0}. Either $i_{j_0} = 0$ and $\mathcal{S}_1 \vdash^* y$ by **Enc0**. Or $i_{j_0} \geq 1$ and we must have that there exist $n_{j_0} \in \mathsf{Nonce}, b \in H, \alpha' \in \{r, g\}$ such that $\mathcal{S}_1 \vdash^* h_b^{\alpha'}(n_{j_0}, y, i_{j_0}, S_{j_0})$, that is $(i_{j_0}, S_{j_0}) < \mathsf{SecDegree}(K, \mathcal{S}_1)$.

Let us conclude that $\mathcal{S}_1'$ satisfies **SecW***. Assume $\mathcal{S}_1' \vdash^W h_a^\alpha(x, y, i, S)$ and $(i, S) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$ with $S \subseteq H, a \in H$. Assume that $\mathcal{S}_1' \vdash^W y$ and let us show $y \in K$. We have $\mathcal{S}_1 \vdash^W h_a^\alpha(x, y, i, S)$ thus property **Sec*** ensures $y \in \mathsf{Key} \cup \mathsf{Nonce}$ and $\mathcal{S}_1 \not\vdash^W y$. We have just proved that $\mathcal{S}_1' \vdash^W y$ implies $\mathcal{S}_1 \vdash^* y$ or $y \in K$ or there exists $\mathcal{S}_1 \vdash^* h_{a'}^{\alpha'}(n, y, i', S')$ such that $(i', S') < \mathsf{SecDegree}(K, \mathcal{S}_1)$. In this last case, Property **SecDegree** on $\mathcal{S}_1$ ensures $i = i'$ and $S = S'$, contradiction. Thus we must have $y \in K$.

We now show that properties **SecW***, **EncW** and **Enc0W** are invariant under application of the rules of the API. Let $\mathcal{S}$ be a state satisfying **SecW***, **EncW** and **Enc0W** and consider a state $\mathcal{S}'$ such that $\mathcal{S} \rightarrow_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$. Let us show that $\mathcal{S}'$ satisfies **SecW***, **EncW** and **Enc0W**

Let us first notice that properties **SecW***, **EncW** and **Enc0W** are clearly invariant under application of the rules of $\mathsf{ATTACKER} \cup \mathsf{CONTROL}$. Indeed, if $\mathcal{S} \rightarrow_{\mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$ then for any term u, we have $S' \vdash^W u$ if and only if $S \vdash^W u$. For the last rule of $\mathsf{CONTROL}$, this is due to the fact that whenever $h_b^\alpha(n, k, i, S) \in \mathcal{S}$ with $b \notin H$ then $\mathcal{S} \vdash^W h_b^\alpha(n, k, i, S)$.

Thus we now assume $\mathcal{S} \rightarrow_{\mathsf{API}} \mathcal{S}'$ and we consider three cases depending on the rule that has been applied.

Secure and Public Generation rule. $a \in S \subseteq \mathsf{Agent}$

$$\begin{array}{ll} \overset{N,K}{\Rightarrow} & P_a(h_a^g(N, K, i, S)) \text{ and } i \in \{1, 2\} \\ \overset{N,K}{\Rightarrow} & P_a(K), P_a(h_a^g(N, K, 0, \mathsf{All})) \end{array}$$

$\mathcal{S}' = \mathcal{S} \cup \{h_a^g(n, k, i, S)\}$ and $i \in \{1, 2\}$ or $\mathcal{S}' = \mathcal{S} \cup \{k, h_a^g(n, k, 0, \mathsf{All})\}$ where n is a fresh nonce and k is a fresh key. Using that only fresh values are added, we easily check that $\mathcal{S}$ satisfies properties **SecW***, **EncW** and **Enc0W** implies $\mathcal{S}'$ satisfies properties **SecW***,

EncW and **Enc0W**.

Encryption rule.

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(m_1), \dots, P_a(m_p) \Rightarrow P_a(\{m'_1, \dots, m'_p\}_{X_k})$$

We have $m_j\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$ and $\mathcal{S}' = \mathcal{S} \cup \{(\{m'_1, \dots, m'_p\}_{X_k})\theta\}$ where the m_j and m'_j are defined as in rule **Encrypt**. Let $m'_j = m''_j, i_j, S_j$. If $a \notin H$ then $\mathcal{S} \vdash^W m'_i\theta$ and $\mathcal{S} \vdash^W X_k\theta$. Thus $\mathcal{S}' \vdash^W u$ if and only if $\mathcal{S} \vdash^W u$ for any term or handle u. We deduce that $\mathcal{S}$ satisfies properties **SecW***, **EncW** and **Enc0W** implies $\mathcal{S}'$ satisfies properties **SecW***, **EncW** and **Enc0W** in the case $a \notin H$.

We now assume $a \in H$. We distinguish between two cases.

- Either $\mathcal{S} \vdash^W X_k\theta$. In that case, since $\mathcal{S} \vdash^W h_a^\alpha(X_n, X_k, i_0, S_0)\theta$, property **SecW*** ensures that $S_0 \not\subseteq H$ or $X_k\theta \in K$ or $(i_0, S_0) < \mathsf{SecDegree}(K, \mathcal{S}_1)$. We deduce that for any $1 \leq i_j \leq 3$, we have $S_j \not\subseteq H$ or $(i_j, S_j) < \mathsf{SecDegree}(K, \mathcal{S}_1)$. Thus $m_j\theta = h_a^\alpha(n_j, k_j, i_j, S_j)$ and $\mathcal{S} \vdash^W k_j$. For any $i_j = 0$, we have $m_j\theta \in \mathcal{S}$. Thus in both cases, we deduce that $\mathcal{S} \vdash^W m'_j\theta$ for any $1 \leq j \leq p$. This implies that $\mathcal{S} \vdash^W (\{m'_1, \dots, m'_p\}_{X_k})\theta$ thus the set of terms deducible from $\mathcal{S}$ is identical to the set of terms deducible from $\mathcal{S}'$. We can therefore deduce that $\mathcal{S}$ satisfies properties **SecW***, **EncW** and **Enc0W** implies $\mathcal{S}'$ satisfies properties **SecW***, **EncW** and **Enc0W**.

- Or $\mathcal{S} \not\vdash^W X_k\theta$. Thus it must be the case that $S_0 \subseteq H$ and $(i_0, S_0) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$ (otherwise $\mathcal{S} \vdash^W h_a^\alpha(X_n, X_k, i_0, S_0)\theta$ implies $\mathcal{S} \vdash^W X_k\theta$). Moreover, applying Lemma 1 Equation A.1 (extended to the deduction relation $\vdash^W$), we get that $\mathcal{S}' \vdash^W u$ iff $\mathcal{S} \vdash^W u$ for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ (*).

  Property **SecW***: Assume $\mathcal{S}' \vdash^W h_a^\alpha(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $a \in H$, $E_1 \subseteq H$, $k_1 \notin K$, and $(l_1, E_1) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$. Then (*) implies $\mathcal{S} \vdash^W h_a^\alpha(n_1, k_1, l_1, E_1)$. Thus Property **SecW*** implies $\mathcal{S} \not\vdash^W k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$. Thus applying (*) again, we get $\mathcal{S}' \not\vdash^W k_1$. We conclude that $\mathcal{S}'$ satisfies Property **SecW***.

  Property **EncW**: Assume $\mathcal{S}' \vdash^W \{u\}_k$ for some $u, k \in \mathsf{Msg}$ and assume $\mathcal{S}' \not\vdash^W k$. If $\mathcal{S} \vdash^W \{u\}_k$, we conclude using the fact that $\mathcal{S}$ satisfies Property **EncW**. Otherwise, we must have $\{u\}_k = \{m'_1, \dots, m'_p\}_{X_k}\theta$. Thus in both cases $\mathcal{S}'$ satisfies Property **EncW**.

  Property **Enc0W**: Assume $\mathcal{S}' \vdash^W \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$ or $S_j \not\subseteq H$. If $\mathcal{S} \vdash^W \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0W**. If $\mathcal{S}' \vdash^W k$ then we of course have $\mathcal{S}' \vdash^W u_j$. Otherwise, by Lemma 1 Equation A.2, we must have $\{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k = (\{m'_1, \dots, m'_p\}_{X_k})\theta$. Thus $S_j = S'_j$ and $m_j\theta = u_j \in \mathcal{S}$ if $i_j = 0$ or $m_j\theta = h(n_j, u_j, i_j, S_j)$ with $S_j \not\subseteq H$. In both cases, we deduce $\mathcal{S}' \vdash^W u_j$ and $\mathcal{S}'$ satisfies Property **Enc0W**.

Decryption rule.

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(\{m_1, \dots, m_p\}_{X_k}), \bigcup_{j \in L} P_a(m'_j) \overset{N_1, \dots, N_k}{\Rightarrow} \bigcup_{j \notin L} P_a(m'_j)$$

If $a \notin H$ then $\mathcal{S} \vdash^W m_i\theta$ and $\mathcal{S} \vdash^W X_k\theta$. Moreover, for any $m'_j\theta$ of the form $h(n_j, u_j, i_j, S_j)$ with corresponding $m_j\theta = u_j, i_j, S_j$, we have $a \in S_0 \subseteq S_j$ thus $S_j \not\subseteq H$ Property **Enc0W** ensures that $\mathcal{S} \vdash^W u_j$. Thus applying Lemma 1 (extended to $\vdash^W$), Thus $\mathcal{S}' \vdash^W u$ if and only if $\mathcal{S} \vdash^W u$ for any term u and for any handle u except if u is a handle and one of the $m'_i\theta$. We deduce that $\mathcal{S}$ satisfies properties **EncW** and **Enc0W** implies $\mathcal{S}'$ satisfies properties **EncW** and **Enc0W**. Moreover, $\mathcal{S} \vdash^W X_k\theta$ implies $S_0 \not\subseteq H$ or $(i_0, S_0) < \mathsf{SecDegree}(K, \mathcal{S}_1)$ or $X_k\theta \in K$ since $\mathcal{S}$ satisfies **SecW***. Thus for any $m_j\theta$ of the form i_j, S_j, u_j we must have $S_j \not\subseteq H$ or $(i_j, S_j) < \mathsf{SecDegree}(K, \mathcal{S}_1)$. Thus $\mathcal{S}'$ satisfies **SecW*** even for the handles introduced by the decryption rule. We deduce $\mathcal{S}'$ satisfies properties **SecW***, **EncW** and **Enc0W**.

We now assume $a \in H$. We have $\{m_1, \dots, m_k\}_{X_k}\theta \in \mathcal{S}$, $h^\alpha_a(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$, $m'_j\theta \in \mathcal{S}$ for any $j \in L$ and $\mathcal{S}'_a = \mathcal{S}_a \cup \{m'_j\theta \mid j \notin L\}$ where the m_i and m'_i are defined as in rule **Encrypt**. Let $m_j = m''_j, i_j, S_j$. For any j such that $i_j = 0$, Property **Enc0W** ensures that $\mathcal{S} \vdash^W m'_j\theta$. For any j such that $i_j \geq 1$, then $m'_j\theta$ is a fresh handle of the form $h(n_k, u_j, i_j, S_j)$. Either $S_j \not\subseteq H$ and Property **Enc0W** ensures that $\mathcal{S} \vdash^W u_j$ or $S_j \subseteq H$. Thus we can deduce from Lemma 1 (extended to $\vdash^W$) that for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$, we have $\mathcal{S}' \vdash^W u$ iff $\mathcal{S} \vdash^W u$ or $u = m'_j\theta$ for some j such that $i_j \geq 1$ (**).

Property **SecW***: Assume $\mathcal{S}' \vdash^W h^{\alpha'}_b(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $a \in H$, $E_1 \subseteq H$, $k_1 \notin K$, $(l_1, E_1) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$. Then (**) implies $\mathcal{S} \vdash^W h^{\alpha'}_b(n_1, k_1, l_1, E_1)$ or $h^{\alpha'}_b(n_1, k_1, l_1, E_1) = m'_j\theta$ for some j such that $i_j \geq 1$. In the first case ($\mathcal{S} \vdash^W h^{\alpha'}_b(n_1, k_1, l_1, E_1)$), Property **SecW*** ensures $\mathcal{S} \not\vdash^W k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^W k_1$. In the second case ($h^{\alpha'}_b(n_1, k_1, l_1, E_1) = m'_j\theta$) then $m_j\theta = k_1$. Thus we must have $S_0 \subseteq H$ (since $S_0 \subseteq E_1$ by construction of the decryption rule) and $(i_0, S_0) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$ (since $(l_1, E_1) \not< \mathsf{SecDegree}(K, \mathcal{S}_1)$). If $X_k\theta \in K$ we would have $(l_1, E_1) < (i_0, S_0) \in \mathsf{SecDegree}(K, \mathcal{S}_1)$. Thus Property **SecW*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^W X_k\theta$. Property **EncW** on $\mathcal{S}$ ensures that there exists $n' \in \mathsf{Nonce}$, $\alpha'' \in \{r, g\}$, $c \in H$ such that $\mathcal{S} \vdash^W h^{\alpha''}_c(n', k_1, l_1, E_1)$. Property **SecW*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^W k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^W k_1$. In both cases we conclude that $\mathcal{S}'$ satisfies Property **SecW***.

Property **EncW**: Assume $\mathcal{S}' \vdash^W \{u\}_k$ and $\mathcal{S}' \not\vdash^W k$ for some $u, k \in \mathsf{Msg}$. We have $\mathcal{S} \not\vdash^W k$. By Lemma 1 and (**), we must have $\mathcal{S} \vdash^W \{u\}_k$. Thus Property **EncW** on $\mathcal{S}$ implies that Property **EncW** holds for $\mathcal{S}'$.

Property **Enc0W**: Assume $\mathcal{S}' \vdash^W \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$ and $i'_j = 0$ or $S'_j \not\subseteq H$. If $\mathcal{S} \vdash^W \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0W**. Otherwise, by Lemma 1 Equation A.2, we must have $\mathcal{S}' \vdash^W k$ thus $\mathcal{S}' \vdash^W u_j$.

Appendix C. Proof of Theorem 3

Let $\mathcal{S}$ be a state satisfying **SecFresh***, **Enc'** and **Enc0'** and consider a state $\mathcal{S}'$ such that $\mathcal{S} \to_{\mathsf{API} \cup \mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$. Let us show that $\mathcal{S}'$ satisfies **SecFresh***, **Enc'** and **Enc0'**

Let us first notice that properties **SecFresh***, **Enc'** and **Enc0'** are clearly invariant under application of the rules of $\mathsf{ATTACKER} \cup \mathsf{CONTROL}$. Indeed, if $\mathcal{S} \rightarrow_{\mathsf{ATTACKER} \cup \mathsf{CONTROL}} \mathcal{S}'$ then for any term u, we have $S' \vdash^* u$ if and only if $S \vdash^* u$. For the last rule of $\mathsf{CONTROL}$, this is due to the fact that whenever $h^\alpha_b(n, k, i, S) \in \mathcal{S}$ with $b \notin H$ then $\mathcal{S} \vdash^* h^\alpha_b(n, k, i, S)$.

Thus we now assume $\mathcal{S} \rightarrow_{\mathsf{API}^r} \mathcal{S}'$ and we consider three cases depending on the rule that has been applied.

Secure and Public Generation rules. $a \in S \subseteq \mathsf{Agent}$

$$\begin{array}{ll} \stackrel{N,K}{\Rightarrow} & P_a(h^g_a(N, K, i, S)) \text{ and } i \in \{1, 2\} \\ \stackrel{N,K}{\Rightarrow} & P_a(K), P_a(h^g_a(N, K, 0, \mathsf{All})) \end{array}$$

$\mathcal{S}' = \mathcal{S} \cup \{h^g_a(n, k, i, S)\}$ and $i \in \{1, 2\}$ or $\mathcal{S}' = \mathcal{S} \cup \{k, h^g_a(n, k, 0, \mathsf{All})\}$, where n is a fresh nonce and k is a fresh key. Using the fact that only fresh values are added, we easily check that $\mathcal{S}$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** implies $\mathcal{S}'$ satisfies properties **SecFresh***, **Enc'** and **Enc0'**.

Encryption rule.

$$P_a(h^\alpha_a(X_n, X_k, i_0, S_0)), P_a(m_1), \ldots, P_a(m_p) \Rightarrow P_a(\{m'_1, \ldots, m'_p\}_{X_k})$$

We have $m_j\theta \in \mathcal{S}$, $h^\alpha_a(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$ and $\mathcal{S}' = \mathcal{S} \cup \{(\{m'_1, \ldots, m'_p\}_{X_k})\theta\}$ where the m_j and m'_j are defined as in rule **Encrypt**. Let $m'_j = m''_j, i_j, S_j$. If $a \notin H$ then $\mathcal{S} \vdash^* m'_i\theta$ and $\mathcal{S} \vdash^* X_k\theta$. Thus $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term or handle u. We deduce that $\mathcal{S}$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** implies $\mathcal{S}'$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** in the case $a \notin H$.

We now assume $a \in H$. We distinguish between two cases.

- Either $\mathcal{S} \vdash^* X_k\theta$. In that case, since $\mathcal{S} \vdash^* h^\alpha_a(X_n, X_k, i_0, S_0)\theta$, property **SecFresh*** ensures that $S_0 \not\subseteq H$. We deduce that for any $1 \leq i_j \leq 3$, we have $S_j \not\subseteq H$ thus $m_j\theta = h^\alpha_a(n_j, k_j, i_j, S_j)$ and $\mathcal{S} \vdash^* k_j$. For any $i_j = 0$, we have $m_j\theta \in \mathcal{S}$. Thus in both cases, we deduce that $\mathcal{S} \vdash^* m'_j\theta$ for any $1 \leq j \leq p$. This implies that $\mathcal{S} \vdash^* (\{m'_1, \ldots, m'_p\}_{X_k})\theta$ thus the set of terms deducible from $\mathcal{S}$ is identical to the set of terms deducible from $\mathcal{S}'$. We deduce that $\mathcal{S}$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** implies $\mathcal{S}'$ satisfies properties **SecFresh***, **Enc'** and **Enc0'**.

- Or $\mathcal{S} \not\vdash^* X_k\theta$. Thus it must be the case that $S_0 \subseteq H$ (otherwise $\mathcal{S} \vdash^* h^\alpha_a(X_n, X_k, i_0, S_0)\theta$ implies $\mathcal{S} \vdash^* X_k\theta$). Moreover, applying Lemma 1 (Equation A.1), we get that $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ (*).

  Property **SecFresh***: Assume $\mathcal{S}' \vdash^* h^\alpha_a(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $a \in H$ and $E_1 \subseteq H$. Then (*) implies $\mathcal{S} \vdash^* h^\alpha_a(n_1, k_1, l_1, E_1)$. Thus Property **SecFresh*** implies $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and moreover $k_1 \in \mathsf{Fresh}$ in case $l_3 \neq 3$. Thus applying (*) again, we get $\mathcal{S}' \not\vdash^* k_1$. We conclude that $\mathcal{S}'$ satisfies Property **SecFresh***.

  Property **Enc'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h^\alpha_a(n, k, i, S)$ with $a \in H$, $S, S_j \subseteq H$ and $i_j \geq 1$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we conclude using

Lemma 1 and the fact that $\mathcal{S}$ satisfies Property **Enc'**. Otherwise, by Lemma 1 ((Equation A.2)), we must have $\{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k = \{m'_1, \dots, m'_p\}_{X_k}\theta$. Thus $S_j = S'_j$ and $m_j\theta = h^\alpha_a(n_j, u_j, i_j, S_j) \in \mathcal{S}$. Thus $\mathcal{S}' \vdash^* h^\alpha_a(n_j, u_j, i_j, S_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**.

Property **Enc0'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$ or $S'_j \not\subseteq H$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0'**. If $\mathcal{S}' \vdash^* k$ then we of course have $\mathcal{S}' \vdash^* u_j$. Otherwise, by Lemma 1 (Equation A.2), we must have $\{i'_1, S'_1, u_1, \dots, i'_p, S'_p, u_p\}_k = (\{m'_1, \dots, m'_p\}_{X_k})\theta$. Thus $S_j = S'_j$ and $m_j\theta = u_j \in \mathcal{S}$ if $i_j = 0$ or $m_j\theta = h(n_j, u_j, i_j, S_j)$ with $S_j \not\subseteq H$. In both cases, we deduce $\mathcal{S}' \vdash^* u_j$ and $\mathcal{S}'$ satisfies Property **Enc0'**.

Decryption rule.

$$P_a(h^\alpha_a(X_n, X_k, i_0, S_0)), P_a(\{m_1, \dots, m_p\}_{X_k}), \bigcup_{j\in L} P_a(m'_j) \overset{N_1,\dots,N_k}{\Rightarrow} \bigcup_{j\notin L} P_a(m'_j)$$

We have $\{m_1, \dots, m_p\}_{X_k}\theta \in \mathcal{S}$, $h^\alpha_a(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$, $m'_j\theta \in \mathcal{S}$ for any $j \in L$ and $\mathcal{S}' = \mathcal{S} \cup \{m'_j\theta \mid j \notin L\}$ where the m_i and m'_i are defined as in rule **Encrypt**. Let $m_j = m''_j, i_j, S_j$. Since the decryption rule belongs to API^r, we must have that if $i_0 = 3$ then L is not empty.

If $a \notin H$ then $\mathcal{S} \vdash^* m_i\theta$ and $\mathcal{S} \vdash^* X_k\theta$. Moreover, for any $m'_j\theta$ of the form $h(n_j, u_j, i_j, S_j)$ with corresponding $m_j\theta = u_j, i_j, S_j$, we have $a \in S_0 \subseteq S_j$ thus $S_j \not\subseteq H$ Property **Enc0** ensures that $\mathcal{S} \vdash^* u_j$. Thus applying Lemma 1, $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term u and for any handle u except if u is a handle and one of the $m'_i\theta$. We deduce that $\mathcal{S}$ satisfies properties **Enc** and **Enc0** implies $\mathcal{S}'$ satisfies properties **Enc** and **Enc0**. Moreover, $\mathcal{S} \vdash^* X_k\theta$ implies $S_0 \not\subseteq H$ since $\mathcal{S}$ satisfies **Sec***. Thus for any $m_j\theta$ of the form i_j, S_j, u_j we must have $S_j \not\subseteq H$. Thus $\mathcal{S}'$ satisfies **Sec*** even for the handles introduced by the decryption rule. We deduce $\mathcal{S}'$ satisfies properties **Sec***, **Enc** and **Enc0**.

We now assume $a \in H$.

Assume first that $S_0 \subseteq H$. If $i_0 \neq 3$ then Property **SecFresh*** ensures that $k \in \mathsf{Fresh}$, which ensures that $\{m_1, \dots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. If $i_0 = 3$, then L is not empty. Let $j_0 \in L$, we have $m_{j_0} = i_{j_0}, S_{j_0}, w_{j_0}$. There exists n_{j_0} such that $h^g_a(n_{j_0}, w_{j_0}, i_{j_0}, S_{j_0}) \in \mathcal{S}$. Then Property **SecFresh*** ensures that $w_{j_0} \in \mathsf{Fresh}$, which ensures that $\{m_1, \dots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. In both cases, if $S_0 \subseteq H$ we can conclude that $\{m_1, \dots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$.

Moreover, for any j such that $i_j = 0$, Property **Enc0'** ensures that $\mathcal{S} \vdash^* m'_j\theta$. For any j such that $i_j \geq 1$, then $m'_j\theta$ is a fresh handle of the form $h(n_k, u_j, i_j, S_j)$. Either $S_j \not\subseteq H$ and Property **Enc0** ensures that $\mathcal{S} \vdash^* u_j$ or $S_j \subseteq H$. Thus we can deduce from Lemma 1 (Equation A.1) that for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$, we have $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ or $u = m'_j\theta$ for some j such that $i_j \geq 1$ (**).

Assume now that $S_0 \not\subseteq H$. Then $\mathcal{S} \vdash^* X_k\theta$ thus for any $1 \leq i \leq p$, $\mathcal{S} \vdash^* m_i$. In particular, for any j such that $i_j = 0$, $\mathcal{S} \vdash^* m'_j\theta$. Thus we deduce that property (**) also holds.

Property **SecFresh***: Assume $\mathcal{S}' \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1,2,3\}$, $b \in H$, $E_1 \subseteq H$. Then (**) implies $\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ or $h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j\theta$ for some j such that $i_j \geq 1$. In the first case ($\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$), Property **SecFresh*** ensures $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and $k_1 \in \mathsf{Fresh}$ is $l_1 \neq 3$. Thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In the second case ($h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j\theta$) then $m_j\theta = k_1$ and $S_0 \subseteq E_1 \subseteq H$ thus $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. Property **Enc'** then ensures that there exists $n' \in \mathsf{Nonce}$, $c \in H$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n', k_1, l_1, E_1)$. Property **SecFresh*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and $k_1 \in \mathsf{Fresh}$ is $l_1 \neq 3$. Thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In both cases we conclude that $\mathcal{S}'$ satisfies Property **SecFresh***.

Property **Enc'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h_b^{\alpha'}(n, k, i, S)$ with $b \in H$, $S, S'_j \subseteq H$ and $i_j \geq 1$. Property **SecFresh*** on $\mathcal{S}'$ ensures that $\mathcal{S}' \not\vdash^* k$. Thus by Lemma 1 (Equation A.2), we must have $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$. Moreover, (**) implies that either $\mathcal{S} \vdash^* h_b^{\alpha'}(n, k, i, S)$ or $h_b^{\alpha'}(n, k, i, S) = m'_j\theta$ for some $l \notin L$. In the first case, we conclude using the fact that $\mathcal{S}$ satisfies Property **Enc'** thus $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k \in \mathcal{S}_0$ or $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and there exists $n_j \in \mathsf{Nonce}$ and $c \in H$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$, which implies $\mathcal{S}' \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**. In the second case, we must have $i \neq 3$ and $S_0 \subseteq S \subseteq H$ thus $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. We have $\{m_1, \ldots, m_k\}_{X_k}\theta \in \mathcal{S}$ and $h_a^{\alpha}(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$. Since $\mathcal{S}$ enjoys Property **Enc'**, we deduce that there exists $n_l \in \mathsf{Nonce}$ and $c \in H$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i_l, S_l)$, that is $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i, S)$. Since $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i \neq 3$, Property **SecFresh*** ensures that $k \in \mathsf{Fresh}$ thus $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k \notin \mathcal{S}_0$. Property **Enc'** on $\mathcal{S}$ thus ensures that $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and that there exists $n'_j \in \mathsf{Nonce}$ and $d \in H$ such that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$. We can deduce that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**.

Property **Enc0'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i'_j = 0$ or $S'_j \not\subseteq H$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0'**. Otherwise, by Lemma 1 (Equation A.2), we must have $\mathcal{S}' \vdash^* k$ thus $\mathcal{S}' \vdash^* u_j$.

Appendix D. Prolog Implementation

```
%% Demonstrates the generation of appropriate API rules from a protocol.

%% Identities of participants
participant(alice).
participant(bob).
participant(server).

% protocol_step/4 specifies the protocol. Usually loaded from a second file,
% but given here in the listing for illustrative purposes
% Carlsson secret key initiator

%A->B:  A, Na
%B->S:  A, Na, B, Nb
%S->B:  {Kab, Nb, A}Kbs, {Na, B, Kab}Kas
%B->A: {Na, B, Kab}Kas, {Na}Kab, N'b
%A->B:  {N'b}Kab

protocol_step(alice,
              [],
```

```
                [nonce(alice,na,0,[alice,bob,server])],
                [agent(alice),nonce(alice,na,0,[alice,bob,server])]).

protocol_step(bob,
                [agent(alice),nonce(alice,na,0,[alice,bob,server])],
                [nonce(bob,nb,0,[alice,bob,server])],
                [agent(alice),nonce(alice,na,0,[alice,bob,server]),
                 agent(bob),nonce(bob,nb,0,[alice,bob,server])]).

protocol_step(server,
                [agent(alice),nonce(alice,na,0,[alice,bob,server]),agent(bob),
                 nonce(bob,nb,0,[alice,bob,server])],
                [key(server,kab,2,[alice,bob,server])],
                [enc([key(server,kab,2,[alice,bob,server]),
                 nonce(bob,nb,0,[alice,bob,server]),agent(alice)],kbs),
                 enc([nonce(alice,na,0,[alice,bob,server]),agent(bob),
                 key(server,kab,2,[alice,bob,server])],kas)]).

protocol_step(bob,
                [enc([key(server,kab,2,[alice,bob,server]),
                 nonce(bob,nb,0,[alice,bob,server]),agent(alice)],kbs),
                 pakcet],
                [nonce(bob,nbb,0,[alice,bob,server])],
                [packet,enc([nonce(alice,na,0,[alice,bob,server])],kab),
                 nonce(bob,nbb,0,[alice,bob,server])]).

protocol_step(alice,
                [enc([nonce(alice,na,0,[alice,bob,server]),agent(bob),
                 key(server,kab,2,[alice,bob,server])],kas),
                 enc([nonce(alice,na,0,[alice,bob,server])],kab),
                 nonce(bob,nbb,0,[alice,bob,server])],
                [],
                [enc([nonce(bob,nbb,0,[alice,bob,server])],kab)]).

protocol_step(bob,
                [enc([nonce(bob,nbb,0,[alice,bob,server])],kab)],
                [],
                []).

%% initial handles

handle(h(alice,kas,3,[alice,server])).
handle(h(bob,kbs,3,[bob,server])).
handle(h(server,kas,3,[alice,server])).
handle(h(server,kbs,3,[bob,server])).

%% for sicstus prolog
:- dynamic command/4.
% handle store
:- dynamic handle/1.

%% for Gnu prolog
%dynamic(command/4). etc

%% predicate to get a new name for a handle

new_handle_name(Agent,Name,Name) :-
        \+ handle(h(Agent,Name,_,_)).

new_handle_name(Agent,Name,Name) :-
        new_handle_name(Agent,prime(Name),prime(Name)).

%%% the algorithm

go(_Protocol) :-
```

```
                go,
                print_commands.

go :-
        findall(_,(
                protocol_step(Played_by,LHS,MID,RHS),
                treat_step(Played_by,LHS,MID,RHS)),_).

print_commands :-
        findall(_,(
                command(Played_by,Commands1,Commands2,Commands3),
                format("~n~n
Command_for_~w_~n~n
Generate_~w_~n~n
Decrypt_~w_~n~n
Encrypt_~w_~n~n",
                        [Played_by,Commands1,Commands2,Commands3])),_).

treat_step(Played_by,LHS,MID,RHS) :-

        generate_rules(Played_by,MID,Commands1),
        decrypt_rules(Played_by,LHS,Commands2), % tests included here
        encrypt_rules(Played_by,RHS,Commands3),
        assert(command(Played_by,Commands1,Commands2,Commands3)),!.

%% Determine generate rules
generate_rules(_,[],[]).

generate_rules(Agent,[nonce(Agent,Name,Level,Set)|Rest],
                        [gen(h(Agent,Name,Level,Set))|OtherCommands]) :-
        assert(handle(h(Agent,Name,Level,Set))),
        generate_rules(Agent,Rest,OtherCommands).

generate_rules(Agent,[key(Agent,Name,Level,Set)|Rest],
                        [gen(h(Agent,Name,Level,Set))|OtherCommands]) :-
        assert(handle(h(Agent,Name,Level,Set))),
        generate_rules(Agent,Rest,OtherCommands).

%% Determine decryption rules
decrypt_rules(_,[],[]).

decrypt_rules(Agent,[enc(List,Key)|Rest],[decrypt([enc(List,Key),
                        h(Agent,Key,Level,Set),Test],RHS)|Commands]) :-
        handle(h(Agent,Key,Level,Set)),
        treat_encrypted_terms(Agent,List,RHS),
        derive_tests(Agent,List,Test),
        decrypt_rules(Agent,Rest,Commands).


decrypt_rules(Agent,[_|Rest],Commands) :-
        decrypt_rules(Agent,Rest,Commands).

%% treat the encrypted terms that are decrypted
treat_encrypted_terms(_,[],[]) :- !.

treat_encrypted_terms(Agent,[agent(Name)|Rest],[agent(Name)|RHS]) :-
                        treat_encrypted_terms(Agent,Rest,RHS).

%forwarding of encrypted packets.

treat_encrypted_terms(Agent,[packet|Rest],[packet|RHS]) :-
                        treat_encrypted_terms(Agent,Rest,RHS).

%% nonce level 0 returned
```

```
treat_encrypted_terms(Agent,[nonce(Agent2,Name,0,S)|Rest],
                                    [nonce(Agent2,Name,0,S)|LHS]) :-
                  treat_encrypted_terms(Agent,Rest,LHS).


%%% nonce level 1 - get a new handle for it

treat_encrypted_terms(Agent,[nonce(_Agent2,Name,1,Set)|Rest],
                                    [h(Agent,Name2,1,Set)|LHS]) :-
                  new_handle_name(Agent,Name,Name2),
                  assert(handle(h(Agent,Name2,1,Set))),
                  treat_encrypted_terms(Agent,Rest,LHS).


%%% key level 2 - get a new handle for it

treat_encrypted_terms(Agent,[key(_Agent2,Name,2,Set)|Rest],
                                    [h(Agent,Name2,2,Set)|LHS]) :-
                  new_handle_name(Agent,Name,Name2),
                  assert(handle(h(Agent,Name2,2,Set))),
                  treat_encrypted_terms(Agent,Rest,LHS).

%% Finally the rules for encryption

encrypt_rules(_Agent,[],[]).

%% only an enc term is of interest

encrypt_rules(Agent,[enc(List,Key)|Rest],
                   [encrypt([h(Agent,Key,L,Set),Terms],
                   [enc(List,Key)])|Rules1]) :-

        handle(h(Agent,Key,L,Set)),
                              % check for nested encryptions
        encrypt_rules(Agent,List,Rules2),
        treat_plaintexts(Agent,List,Terms),
        encrypt_rules(Agent,Rest,Rules3),
        append(Rules3,Rules2,Rules1).

%% other stuff (this can also triggered if the encryption
%% is with a key we don't have,
%% i.e. we forward a secret packet

encrypt_rules(A,[_|Rest],RHS) :-
        encrypt_rules(A,Rest,RHS).

%%%%% treat the things that must be encrypted

treat_plaintexts(_Agent,[],[]).

%  agent name

treat_plaintexts(Agent,[agent(Name)|Rest],[agent(Name)|RHS]) :-
        treat_plaintexts(Agent,Rest,RHS).

%  a nested encryption (it will have its own rule)

treat_plaintexts(Agent,[enc(_,_)|Rest],[nested_encryption|RHS]) :-
        treat_plaintexts(Agent,Rest,RHS).

% one of my nonces, 0 or 1

treat_plaintexts(Agent,[nonce(Agent,Name,L,Set)|Rest],
                          [h(Agent,Name,L,Set)|RHS]) :-
        handle(h(Agent,Name,L,Set)),
```

```
        treat_plaintexts(Agent,Rest,RHS).

%% someone elses nonce level 0 (i.e. public term)

treat_plaintexts(Agent,[nonce(Agent2,Name,0,Set)|Rest],
                          [nonce(Agent2,Name,0,Set)|RHS]) :-
        Agent2 \== Agent,
        treat_plaintexts(Agent,Rest,RHS).

% key

treat_plaintexts(Agent,[key(Agent,Name,L,Set)|Rest],
                          [h(Agent,Name,L,Set)|RHS]) :-

        handle(h(Agent,Name,L,Set)),
        treat_plaintexts(Agent,Rest,RHS).


%% derive tests if necessary, otherwise return the empty list
%% if we find something of level 2, we require
%% a test on something we generated of level 1

derive_tests(Agent,List,Tests) :-
        derive_tests(Agent,List,Tests,List).

derive_tests(_,[],[],_).

derive_tests(Agent,[key(_Agent2,_Name,2,_Set)|_Rest],[test(Handle)],List) :-
        find_test_nonce(Agent,Handle,List).

derive_tests(Agent,[_|Rest],Test,List) :-
        derive_tests(Agent,Rest,Test,List).

find_test_nonce(_,warning(missing_test),[]).

find_test_nonce(Agent,h(Agent,Name,L,Set),[nonce(Agent,Name,L,Set)|_]).

find_test_nonce(Agent,Handle,[_|List]) :-
        find_test_nonce(Agent,Handle,List).
```