A Composable Computational Soundness Notion

Véronique Cortier
LORIA - CNRS
Nancy, France
cortier@loria.fr

Bogdan Warinschi
University of Bristol
Bristol, United Kingdom
bogdan@cs.bris.ac.uk

ABSTRACT

Computational soundness results show that under certain conditions it is possible to conclude computational security whenever symbolic security holds. Unfortunately, each soundness result is usually established for some set of cryptographic primitives and extending the result to encompass new primitives typically requires redoing most of the work. In this paper we suggest a way of getting around this problem.

We propose a notion of computational soundness that we term *deduction soundness*. As for other soundness notions, our definition captures the idea that a computational adversary does not have any more power than a symbolic adversary. However, a key aspect of deduction soundness is that it considers, intrinsically, the use of the primitives in the presence of functions specified by the adversary. As a consequence, the resulting notion is amenable to modular extensions. We prove that a deduction sound implementation of some arbitrary primitives can be extended to include asymmetric encryption and public data-structures (e.g. pairings or list), without repeating the original proof effort.

Furthermore, our notion of soundness concerns cryptographic primitives in a way that is independent of any protocol specification language. Nonetheless, we show that deduction soundness leads to computational soundness for languages (or protocols) that satisfy a so called *commutation property*.

Categories and Subject Descriptors

F.0 [**Theory of Computation**]: General

General Terms

Security, Theory

Keywords

Computational soundness, Composability

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

Security protocols are programs that aim at securing communications over public (insecure) networks. Their design is notoriously error prone and multiple approaches have been devised for rigorous protocol analysis. These techniques fall under one of two categories. Symbolic models ignore most of the cryptographic details using abstract structures like terms (a variant of labeled finite trees). What an adversary can compute out of a set of messages is then typically captured by a symbolic deduction system (that states e.g. that the adversary can decrypt whenever he has the corresponding decryption key). This approach leads to simple models that allow the development of fully automatic decision procedures for security (e.g. [19, 2]). The second category employs computational models. These models are more accurate: they consider a lower level of abstraction and powerful (probabilistic) polynomial time adversaries. Here security does not rely on an axiomatization of the basic primitives, but rather on the difficulty of solving various computational tasks (e.g. factoring or taking discrete logarithms). Proofs in the computational model thus imply stronger guarantees than those in the symbolic models but devising and verifying such proofs is tedious. Even for moderately-sized protocols proofs become extremely long, difficult, and error prone.

An important research direction [7, 1, 5, 4, 20, 13] aims to bridge the gap between the symbolic and the cryptographic approaches via computational soundness results. These results typically show that under reasonable standard cryptographic assumptions, proofs of security with respect to symbolic models directly imply security with respect to the more detailed computational ones. This is a promising approach that allows to obtain strong guarantees while benefiting from the simplicity of symbolic models.

Soundness results have been established for various primitives such as concatenation, asymmetric encryption [5, 13], symmetric encryption [9, 18], signatures [7, 13], hash functions [17, 11], zero-knowledge proofs [6], and non-malleable commitments [14]. See the recent survey [12] for an extensive list. However, each result is dedicated to a small subset of these primitives, typically for a few primitives at a time.

Each of the above results typically considers the security of a few primitives at a time and it is likely the techniques can be extended to cope with new primitives (as long as sound axiomatizations for such primitives are possible). However, the approach of sequentially extending existing results with new primitives has an important drawback. Each new extension requires duplicating large amounts of work and this renders a "unifying" soundness result difficult to obtain. What

is missing is a more modular approach where it is possible to study the soundness of various axiomatizations in isolation and still conclude that the abstraction remains sound when the primitives are used together.

For existent approaches to computational soundness such a result is probably too optimistic. An intuitive counterexample is as follows. Consider some simple programming language that involves only pairing, nonces, and asymmetric encryption. A soundness result for such a language would require that the encryption scheme be IND-CCA secure, but only when used to encrypt bitstrings that arise naturally in the execution of the protocols (i.e. pairs, nonces, ciphertexts). When applied to other bitstrings (say to digital signatures) encryption can be completely insecure (e.g. the identity function) without affecting the soundness result. The extensibility now becomes problematic: adding to the specification language digital signatures would lead to an unsound axiomatization. Protocols may still be symbolically secure but their implementation would be insecure due to the use of an insecure encryption scheme. The example can be easily adapted to show that it is equally problematic to add even a simple datastructures (like lists) in a modular way to existing computational soundness. In both cases the source of the problem is the same: the starting soundness result for encryption does not account for other primitives (or data structures) but those in the programming language under consideration. One can cope with this problem by changing the axiomatization for encryption to reflect its insecurity when encrypting signatures, but this direction is ad-hoc and not modular. More structured approaches to computational soundness that use general compositional frameworks (e.g. [5, 8, 18]) alleviate the scalability problems, but do not yield easily extendable frameworks.

Our results

In this paper we propose and study a computational soundness notion that is amenable to modular extensions. This notion, which we term *deduction soundness* reflects the idea that a symbolic deduction system soundly abstracts the security of the computational implementation for a set of primitives, even when these primitives are used in the presence of other functions for which the implementation is provided by the adversary. This should allow to establish soundness of an implementation, even when additional primitives are around: adding new primitives essentially boils down to fixing the implementation of some of the unspecified functions (as opposed to letting the adversary provide their implementation).

Importantly, in the deduction soundness game, the adversary is only allowed to provide implementations that are *transparent*, which simply means that the implementations must be efficiently invertible. Nonetheless, we confirm that the resulting notion enjoys modular extensibility. We first show that deduction soundness can be extended, essentially for free, with arbitrary data-structures. This is perhaps not surprising as public data structures are given by transparent functions. More interestingly however, we show that deduction soundness can also be extended to asymmetric encryption. The intuition is that encryption under keys for which the adversary knows the corresponding decryption key is essentially a transparent function whereas for other keys encryption can be modelled via transparent constants. We expect that similar extensions can be proved for hash functions (modelled as random oracles) and symmetric encryption.

An important question is whether transparent function can be used to model any other cryptographic primitive. The answer is unfortunately "no". The axioms that define transparent functions only talk about the secrecy (or rather non-secrecy) of their arguments and thus they provide no authentication guarantees. Extensions of our notion with authentication primitives (e.g. signatures and message authentication codes) needs to account for the possible more intricate interaction of the axioms with those for signatures and message authentication codes.

A second aspect of deduction soundness is that it is solely concerned with the implementation of the primitives, and is independent of any protocol structure or programming language where the primitives may be used. The idea to focus only on the axiomatization of individual primitives while disregarding the protocols where the primitives are used can be traced from the work of Herzog [15], through to that of Backes, Pfitzmann, and Waidner [5], and to the more recent CoSP framework of Backes, Hofheinz, and Unruh [3]. As in these latter works we show that soundness (in the form of a mapping lemma) for protocol specification languages can be recovered using our notion. In turn this allows for the translation of trace properties from symbolic models to computational ones.

2. MODELS

In this section we introduce some notations and set our abstract and concrete models. Throughout we assume familiarity of basic cryptographic security notion like indistinguishability under chosen cipher attacks(IND-CCA) for asymmetric encryption schemes, existential unforgeability under chosen message attacks for digital signature and message authentication codes (EU-CMA).

2.1 Abstract algebras

Our abstract models—called *abstract algebras*—consist of term algebras defined on a first-order signature with sorts.

Specifically a *signature* $(\mathcal{S}, \mathcal{F})$ consists of a set of *sorts* $S = \{s, s_1 \ldots\}$, a set of labels $\mathsf{labels} = \mathsf{labelsH} \cup \mathsf{labelsA}$, and a set of *symbols* $\mathcal{F} = \{f, f_1 \ldots\}$ together with arities of the form $\mathrm{ar}(f) = s_1 \times \ldots \times s_k \to s$, $k \geq 0$. Symbols that take $k = 0$ arguments are called *constants*; their arity is simply written s. We fix an infinite set of *variables* $\mathcal{X} = \{x, y \ldots\}$. We assume that variables are given with sorts. The set of *terms of sort* s is defined inductively by

t	::=		term of sort s
	$\mid$	x	variable x of sort s
	$\mid$	$f^l(t_1, \ldots, t_k)$	application of symbol $f \in \mathcal{F}$

where for the last case, we further require that $l \in \mathcal{L}$, t_i is a term of some sort s_i and $\mathrm{ar}(f) = s_1 \times \ldots \times s_k \to s$.

Intuitively, for names, we use (randomized) constants. For example, assume that $n \in \mathcal{F}$ is a constant. Then usual nonces can be represented by $n^{r_1}, n^{r_2}, \ldots$ where $r_1, r_2 \in \mathcal{L}$ are labels. Labels in labelsH will be used when the function has been applied by an honest agent (thus the randomness has been honestly generated) whereas labels in labelsA will be used when the randomness has been generated by the adversary. Often when the label for a functional symbol is clear from the context (e.g. when there is only one label

that suits a particular functional symbol) we may omit this label.

We always assume a supersort term containing all other sorts. We also assume a constant $g_s \in \mathcal{F}$ for any $s \in \mathcal{S}$ that will be used for representing garbage of sort s. Garbage will typically be the terms associated to bit-strings produced by the adversary and which cannot be parsed as a meaningful term. Therefore we require in what follows that for any term g_s^l, the randomness l has to be adversarial: $l \in \mathsf{labelsA}$. As usual, we write $\mathrm{var}(t)$ for the set of variables occurring in t. A term is *ground* or *closed* iff it has no variables. The set of terms is denoted by Term.

Substitutions are written $\sigma = \{x_1 = t_1, \ldots, x_n = t_n\}$ with $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. We only consider *well-sorted* substitutions, that is substitutions $\sigma = \{x_1 = t_1, \ldots, x_n = t_n\}$ for which x_i and t_i have the same sort. σ is *closed* iff all of the t_i are closed. We extend the notation var(.) from terms to substitutions in the obvious way. The application of a substitution σ to a term t is written $\sigma(t) = t\sigma$.

Symbols in $\mathcal{F}$ are intended to model cryptographic primitives, including generation of random data like e.g. nonces or keys. Identities will be typically represented by constants. The term algebra is equipped with a *deduction* relation $\vdash \subseteq 2^{\mathsf{Term}} \times \mathsf{Term}$ that models the information available to a formal adversary. $S \vdash m$ means that a formal adversary can build m out of S, where m is a term and S a set of terms. We say that m is *deducible* from S. Deduction relations are typically defined through deductions rules systems.

Definition 1. *A* deduction system $\mathcal{D}$ *is a set of rules* $\frac{u_1 \quad \cdots \quad u_k}{u}$ *such that* $u_1, \ldots, u_k, u \in \mathsf{Term}$. *The deduction relation* $\vdash_{\mathcal{D}} \subseteq 2^{\mathsf{Term}} \times \mathsf{Term}$ *associated to* $\mathcal{D}$ *is the smallest relation* S *satisfying:*

- *for any* $t \in S$, $S \vdash_{\mathcal{D}} t$;
- *If* $S \vdash_{\mathcal{D}} u_1\theta$, $\ldots S \vdash_{\mathcal{D}} u_k\theta$ *for some substitution* θ *and* $\frac{u_1 \quad \cdots \quad u_k}{u} \in \mathcal{D}$ *then* $S \vdash_{\mathcal{D}} u\theta$.

We may omit the subscript $\mathcal{D}$ in $\vdash_{\mathcal{D}}$ when it is clear from the context.

2.1.1 Specification for signatures and MACs

For example, digital signatures and message authentication codes could be formalized using algebras with the signature Σ_{sm} defined as follows. The set of sorts are given by $\mathcal{S}_{sm} = \{\mathsf{id}, \mathsf{spair}, \mathsf{vkey}, \mathsf{skey}, \mathsf{sigs}, \mathsf{macs}, \mathsf{mkey}\}$ and the set of operations $\mathcal{F}_{sm}$ are given by:

$\mathsf{id}_i :\rightarrow \mathsf{id}$ for $i \in \mathbb{N}$

$\mathsf{skeypair} : \mathsf{id} \rightarrow \mathsf{spair}$ $\quad$ $\mathsf{mackey} : \mathsf{id} \times \mathsf{id} \rightarrow \mathsf{mkey}$

$\mathsf{vk} : \mathsf{spair} \rightarrow \mathsf{vkey}$ $\quad$ $\mathsf{sk} : \mathsf{spair} \rightarrow \mathsf{skey}$

$\mathsf{sign} : \mathsf{skey} \times \mathsf{term} \rightarrow \mathsf{sigs}$ $\quad$ $\mathsf{mac} : \mathsf{mkey} \times \mathsf{term} \rightarrow \mathsf{macs}$

Abusing notation, we often write $\mathsf{sk}(id)$ for $\mathsf{sk}(\mathsf{skeypair}(id))$, write $\mathsf{vk}(id)$ for $\mathsf{vk}(\mathsf{skeypair}(id))$ and write mackey_{ij} for $\mathsf{mackey}(id_i, id_j)$. Also, recall that by an earlier assumption the signature also contains constant garbage symbols of each sort.

The deduction system $\mathcal{D}_{\mathsf{sm}}$ that models the security of signatures and macs is given by rules in Figure 1. The first line specifies that the adversary can recover the verification key of any party, and that it can produce garbage of any type. The next two lines specify, for signature and macs, respectively, that signatures and macs always reveal the message that is being authenticated (even when produced honestly), and that given the secret key the adversary can produce signatures and macs.

$$\frac{}{\mathsf{vk}(x)} \qquad \frac{}{g_s^l}$$

$$\frac{\mathsf{sign}^l(\mathsf{sk}(x), t)}{t} \qquad \frac{\mathsf{sk}(x) \quad t}{\mathsf{sign}^{l_A}(\mathsf{sk}(x), t)}$$

$$\frac{\mathsf{mac}^l(\mathsf{mkey}(x, y), t)}{t} \qquad \frac{\mathsf{mackey}(x, y) \quad t}{\mathsf{mac}^{l_A}(\mathsf{mackey}(x, y), t)}$$

where $x, y \in \mathsf{id}, s \in \mathcal{S}, t \in \mathsf{term}, l \in \mathsf{labels}, l_A \in \mathsf{labelsA}$.

Figure 1: Deduction rules for signatures and macs.

2.2 Concrete implementations for algebras

If A is a randomized algorithm we write $y \leftarrow A(x; r)$ for the process of obtaining y by running A on x with random coins r. We assume a non-empty set of bit-strings $[\![s]\!] \subseteq \{0,1\}^*$ for each sort $s \in \mathcal{S}$. For the supersort term , we assume $[\![\mathsf{term}]\!] = \{0,1\}^*$. We now give terms a concrete semantics, parametrized by:

- a security parameter η,
- an $(\mathcal{S}, \mathcal{F})$-*concrete implementation* of the primitives, given by some Turing Machine M such that for any $f \in \mathcal{F}$, with $\mathrm{ar}(f) = s_1 \times \ldots \times s_k \rightarrow s$,

$$(M\ f) : [\![s_1]\!] \times \cdots \times [\![s_k]\!] \times \{0,1\}^\eta \rightarrow [\![s]\!]$$

The concrete implementation is used in the game that defines computational soundness. In particular, the implementation determines how to parse bitstrings as terms and how to generate the bitstrings that correspond to terms. Next we give some specifics related to these two points.

2.2.1 Assignation sets

The concrete interpretation of a symbolic term is usually obtained by replacing each symbolic function in the term with its concrete implementation. In cryptographic applications functions are often randomized and the same random coins may occur in different places within the same term. This is the case for instance when the same nonce occurs twice in the same term. We record this information via *assignation sets*. Such a set L is simply a set of couples (c, U) where c is a bitstring and U is a list of terms. The idea is that the terms in U are the possible interpretations of the bitstring c. In particular, each such U is either of the form $[g^{l(c)}]$ or of the form $[f^l(g_1, \ldots, g_k); g^{l(c)}]$ where $g^{l(c)}, g_1, \ldots, g_k$ are garbage symbols. The two different forms for U corresponds to the following two situations, respectively. In the first case, c is interpreted as a garbage (i.e. we do not have any special information on it, except possibly its sort). In the second case c is interpreted as $f^l(g_1, \ldots, g_k)$ meaning that c is interpreted as the application of f^l to the interpretation of $g_1, \ldots, g_k$. In the second case, for convenience, we store the two possible interpretations of c which usually are obtained through successive refinements. We may write $(b, u) \in L$ instead of $(b, [u]) \in L$ when $[u]$ is a list with a single element. We say that an assignation set L is

one-to-one and onto if for any $(c, U_1), (c, U_2) \in L$, we have $U_1 = U_2$ and for any $(c_1, U), (c_2, U) \in L$, we have $c_1 = c_2$.

An assignation set L defines the substitution θ_L *associated to an assignation set* L. This is simply the union of the substitutions $g \mapsto f^l(g_1, \ldots, g_k)$ for any $(c, [f^l(g_1, \ldots, g_k); g]) \in L$. For this notion to be well-defined, the substitution needs to be loop free, and in what follows we will ensure that this property is satisfied. A *partial substitution associated to* L is a substitution associated to some assignation set $L' \subseteq L$.

2.2.2 Generating function.

Given an assignation set L we define a generating function that associates a concrete semantics for terms (given the terms already interpreted in L). For reasons that will be clear later, we are only interested in giving semantics to terms labeled by an honest label.

The generation function uses a random tape $\mathcal{R}$ that essentially implements a mapping $\mathcal{R} : \mathsf{Term} \to \{0,1\}^\eta$, mapping a term to random coins used by the randomized operations in that term. Given a closed term t and an assignation set L, $\mathsf{generate}(t, L)$ is defined as follows. We have that t is of the form $f^l(t_1, \ldots, t_n)$ (with possibly $n = 0$).

- If there is $(c, u :: U) \in L$ with $t = u\theta$ for some θ partial substitution associated to L, then return (c, L).
- Otherwise, if $l \in \mathsf{labelsA}$ then the request is rejected ($\mathsf{generate}$ is undefined).
  Otherwise, let $r = \mathcal{R}(f^l(t_1, \ldots, t_n))$, let $(c_i, L) = \mathsf{generate}(t_i, L)$, $1 \leq i \leq n$ and let $c = (M\ f)(c_1, \ldots, c_n, r)$. Then the function returns $(c, L \cup \{(c, [f^l(g^{l(c_1)}, \ldots, g^{l(c_n)}); g^l])\})$.

Note that $\mathsf{generate}(t, L)$ not only returns a bitstring c associated to t but also updates L (to remember, for example, the value associated to t). Note also that $\mathsf{generate}$ depends on the concrete implementation M and the random tape $\mathcal{R}$. When needed, we show explicitly this dependency, but in general we avoid it for readability.

2.2.3 Parsing function

Conversely, we define a function to convert bitstring into terms. A *parsing function* for a $(\mathcal{S}, \mathcal{F})$-concrete implementation M is a function that takes as input a bitstring c and a (one-to-one and onto) assignation set L and returns a term t and a (one-to-one and onto) assignation set that extends L. We impose a particular structure for the function parse. This assumption allows to later extend the parse function associated to an implementation when other primitives are added to the implementation. More precisely, the parse function uses an auxiliary function $\mathsf{saturate}$ (which does the actual parsing). Formally, we assume that the parse function is defined as follows:

$\mathsf{parse}(c, L)$:
If there exists $(c, t :: U) \in L$
  then output $t\theta_L$;
  else let $L = \mathsf{saturate}(L \cup \{(c, [g^{l(c)}])\})$;
    (we assign a new garbage symbol to c with label $l(c) \in \mathsf{labelsA}$)
  let $(c, t :: U) \in L$, output $t\theta_L$;

The $\mathsf{saturate}$ function takes as input an assignation set and outputs an assignation set. Intuitively, the saturate function tries to interpret as a term any bitstring left uninterpreted (to which the current interpretation is set to garbage), given the information present in the assignation set. Notice that when $\mathsf{saturate}$ is called, a new bitstring that is interpreted as garbage had just been added to the assignation set. While interpreting such a bitstring, new bitstrings without an interpretation can be discovered and these are added to the assignation set. The process is repeated until the assignation set does not change.

The exact definition of $\mathsf{saturate}$ function is left unspecified, as it depends on the particular implementation. However, we demand two properties from any $\mathsf{saturate}$ functions used to define parsing. First, if $\mathsf{saturate}$ adds a new bitstring to the assignation set then, the term associated to this bitstring is by default a garbage symbol with adversarial label. Formally, we assume that for any L_1, L_2 such that $L_2 = \mathsf{saturate}(L_1)$ then for any $(b, f^l(g_1, \ldots, g_k) :: U) \in L_2$ if $l \in \mathsf{labelsH}$ then $(b, f^l(g_1, \ldots, g_k) :: U) \in L_1$: no new term with honest labels are introduced. The second assumption is that the order in which the function processes an assignation set does not matter.

$$\mathsf{saturate}(\mathsf{saturate}(L_1) \cup L_2) = \mathsf{saturate}(L_1 \cup L_2)$$

The examples that we give later in the paper satisfy these properties.

2.2.4 Implementation for signatures and MACs

For the implementation that we use for our running example, we assume an efficiently computable, efficiently invertible, one-to-one encoding function $\langle \_ \rangle$ that takes a tuple of bitstrings as input (separated by commas) and returns an encoding of that tuple. If A is a set of bitstrings, and x is a bitstring, then by a slight abuse, we write $\langle A, x \rangle$ for the set $\{\langle a, x \rangle \mid a \in A\}$, and we extend this notation in the obvious way to the case when encode receives several arguments as input. The concrete support sets for every sort s of the signature Σ_{sm} is given by $[\![s]\!] = \langle \{0,1\}^*, s \rangle$, that is any element of the support set is an encoding of a bit-string followed by the name of the support set. The concrete implementation M_{sm} (which we define below) depends on a digital signature scheme $\mathsf{DigSig} = (\mathcal{K}, \mathcal{S}, \mathcal{V})$ and a message authentication code $\mathsf{Mac} = (\mathcal{MK}, \mathcal{T}, \mathcal{MV})$. We assume that from any secret key sk produced by the key generation algorithm of the signature, one can immediately recover its associated verification key vk. Below, (and elsewhere) we write $[\![f]\!]$ for the algorithm that gives the concrete semantics for operation f. We assume some public (efficiently computable) mapping that maps each constant id_i of the signature to some bitstring id_i.

- $(M_{\mathsf{sm}}\ \mathsf{id}_i)$: output $\langle id_i, \mathsf{id} \rangle$
- $(M_{\mathsf{sm}}\ \mathsf{skeypair})$: is $\mathcal{K}(\eta)$ (to the output of which we tag spair)
- $(M_{\mathsf{sm}}\ \mathsf{sk})(s)$: parse s as $\langle (sk, vk), \mathsf{spair} \rangle$ and outputs $\langle sk, \mathsf{skey} \rangle$.
- $(M_{\mathsf{sm}}\ \mathsf{vk})(s)$: parse s as $\langle (sk, vk), \mathsf{spair} \rangle$ and outputs $\langle vk, \mathsf{vkey} \rangle$
- $(M_{\mathsf{sm}}\ \mathsf{sign})(s_1, s_2, r)$: parse s_1 as $\langle sk, \mathsf{skey} \rangle$, calculate $\sigma \leftarrow \mathcal{S}(sk, s_2; r)$, output $\langle \sigma, s_2, vk, \mathsf{sigs} \rangle$, where vk is the verification key associated to sk.

- (M_{sm} mackey): is just $\mathcal{MK}(\eta)$ (to the output of which we tag mackey)

- (M_{sm} mac)(s_1, s_2): parse s_1 as $\langle k, \mathsf{mkey}\rangle$ and output $\langle s_2, \mathcal{T}(k, s_2), \mathsf{macs}\rangle$.

Next, we give the parsing algorithm (that is we specify the function $\mathsf{saturate}_{\mathsf{sm}}$) for the implementation $\mathcal{M}_{\mathsf{sm}}$.

$\mathsf{saturate}_{\mathsf{sm}}(L)$
let $L' = L$
repeat
 let $L = L'$
 for all $(c, [g^{l(c)}]) \in L$ s.t. c is of the form $\langle m, t\rangle$ with $t \in \mathcal{S}_{sm}$
  if $t = \mathsf{id}$ then if $m = id_i$ for some i
   then $L \leftarrow L \setminus \{(c, [g^{l(c)}])\} \cup \{(c, [\mathsf{id}_i, g^{l(c)}])\}$
  if $t = \mathsf{vkey}$ then $L \leftarrow L \setminus \{(c, [g^{l(c)}])\} \cup \{(c, [g^{l(c)}_{\mathsf{vkey}}, g^{l(c)}])\}$
  if $t = \mathsf{skey}$ then $L \leftarrow L \setminus \{(c, [g^{l(c)}])\} \cup \{(c, [g^{l(c)}_{\mathsf{skey}}, g^{l(c)}])\}$
  if $t = \mathsf{sigs}$ and $m = s_1, s_2, s_3$ and there is a such that
   $(s_3, \mathsf{vk}(a) :: U) \in L)$ and $\mathcal{V}(s_3, (s_2, s_1)) = \mathsf{true}$
    then $L \leftarrow L \setminus \{(c, [g^{l(c)}])\} \cup$
     $\{(c, [\mathsf{sign}^{l(c)}(\mathsf{sk}(a), g^{l(s_2)}); g^{l(c)}]), (s_2, g^{l(s_2)})\}$
  if $t = \mathsf{macs}$ and $m = s_2, s_3$ and there exist some a, b
   such that $(s_1, \mathsf{mackey}(a, b) :: U) \in L$ and
   $\mathcal{MV}(s_1, s_2, s_3) = \mathsf{true}$
    then $L \leftarrow L \setminus \{(c, [g^{l(c)}])\} \cup$
   $\{(c, [\mathsf{mac}^{l(c)}(\mathsf{mackey}(a, b), g^{l(s_2)}); g^{l(c)}]), (s_2, g^{l(s_2)})\})$
until $L = L'$
return L

As explained earlier the function saturate attempts to parse all bitstrings in L that are currently interpreted as garbage symbols. For each such occurrence, if the bitstring does not have associated a type, then nothing happens (i.e. the associated term remains a garbage constant). Otherwise the procedure attempts to parse the bitstring depending on the type of the bitstring. For example, when the type is sigs (that is the bitstring is supposedly a signature) the procedure determines that this is a valid signature on some message s_2 for the public key of some party. If this is so, a new interpretation for c is added to the assignation list, as a signature on some new garbage symbol associated to s_2, and $(s_2, g^{l(s_2)})$ is added to the assignation list (to be parsed in the next pass over the repeat loop).

2.3 Transparent implementation

Typical primitives that are usually considered in soundness results include encryption, signature scheme, hash function, etc. We define and study soundness of such primitives when they are used together with a class of functions which we call *transparent* functions.

Intuitively, such functions are efficiently invertible, and the type of their output can be efficiently determined. An example of such functions are data structures (i.e. pairs, lists, XML documents etc.). Formally, a transparent implementation of a signature $(\mathcal{S}, \mathcal{F})$ is a polynomial-time Turing machine M such that for all $f \in \mathcal{F}$, with $\mathrm{ar}(f) = s_1 \times \ldots \times s_k \rightarrow s$,

$$(M\ \mathsf{evaluate}\ f) : [\![s_1]\!] \times \cdots \times [\![s_k]\!] \times \{0,1\}^\eta \rightarrow [\![s]\!]$$
$$(M\ \mathsf{type}) : \{0,1\}^* \rightarrow \mathcal{F} \cup \{\bot\}$$
$$(M\ \mathsf{proj}\ f\ i) : \{0,1\}^* \rightarrow \{0,1\}^* \cup \{\bot\}$$

such that for any $m_i \in [\![s_i]\!]$, $1 \le i \le k$, for any r,

$$(M\ \mathsf{proj}\ f\ i)((M\ \mathsf{evaluate}\ f)(m_1, \ldots, m_k, r)) = m_i$$
$$(M\ \mathsf{type})((M\ \mathsf{evaluate}\ f)(m_1, \ldots, m_k, r)) = f$$

As hinted above, our notion of soundness does not consider primitives when used stand-alone, but considers their use together with a (infinite, but countable) set of transparent functions.

We next explain how to extend, generically, an algebraic signature and its associated deduction system with transparent functions. Furthermore, we explain how to extend the parsing algorithm associated to an implementation when one adds an implementation for transparent functions.

Let $\Sigma_1 = (\mathcal{S}_1, \mathcal{F}_1)$ be a signature and $\mathcal{D}_{\Sigma_1}$ be a set of deduction rules inducing a deduction system such as the deduction system defined in Example 2.1.1. Let $\Sigma_{\mathsf{tran}} = (\mathcal{S}_{\mathsf{tran}}, \mathcal{F}_{\mathsf{tran}})$ be a signature representing a set of transparent functions. Then we define its associated deduction system $\mathcal{D}_{\mathsf{tran}(\Sigma_{\mathsf{tran}})}$ as follows.

$$\mathcal{D}_{\mathsf{tran}(\Sigma_{\mathsf{tran}})} = \left\{ \begin{array}{l} \frac{t_1 \ \cdots \ t_k}{f^l(t_1,\ldots,t_k)}\ l \in \mathsf{labelsA}, f \in \mathcal{F}_{\mathsf{tran}} \\ \\ \frac{f^l(t_1,\ldots,t_k)}{t_i}\ 1 \le i \le k, l \in \mathsf{labels}, f \in \mathcal{F}_{\mathsf{tran}} \end{array} \right\}$$

The deduction relation $\vdash_{\Sigma_1 \cup \Sigma_{\mathsf{tran}}}$ is the deduction relation induced by $\mathcal{D}_{\Sigma_1} \cup \mathcal{D}_{\mathsf{tran}(\Sigma_{\mathsf{tran}})}$.

Given a parsing algorithm parse (that is, the saturate algorithm) for signature Σ_1 and given an implementation M for the transparent functions in $\mathcal{F}_{\mathsf{tran}}$, we define the algorithm $\mathsf{parse}^{\mathsf{tran}}(M)$. This algorithm parses bitstrings as terms over $\mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}}$. In Figure 2 we specify the function $\mathsf{saturate}^{\mathsf{tran}}$ (that defines $\mathsf{parse}^{\mathsf{tran}}$). Notice that we will from now on omit the dependency of this extension on M, since M will always be clear from the context. The extension is (necessarily) a bit technical: Here we are attempting to parse a bitstring as a term over the functions in $\mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}}$, but have only black-box access to the parsing (saturate) algorithm corresponding to $\mathcal{F}_1$. This algorithm may classify as garbage strings that can potentially be successfully interpreted as terms over $\mathcal{F}_2$. We need therefore to maintain the link between the interpretation of the same bitstring as garbage with respect to $\mathcal{F}_1$, but as some term with respect to $\mathcal{F}_{\mathsf{tran}}$. For this we introduce a mechanism which uses an auxiliary function collapse. Given two signatures Σ_1, Σ_2, the collapse function takes as input an assignation set L and replaces any subterm with its head symbol in Σ_2 with a fresh garbage of the same sort. Since the labels of the term and of the garbage symbol are the same (i.e. the bitstring to which they correspond) the result is simply a substitution that replaces garbage symbols in Σ_1 with terms in Σ_2.

Function $\mathsf{collapse}_{\Sigma_1,\Sigma_2}(L)$
 Let $\mathsf{Mem} = \emptyset$; (memorizes garbage symbols and their associated terms)
 for any $f^l(t_1, \ldots, t_k)$ occurring as subterm in u,
  where $(c, u) \in L$ and $f \in \Sigma_2$ is of sort $s_1 \times \cdots \times s_k \rightarrow s$
  let $\mathsf{Mem} = \mathsf{Mem} \cup \{(g_s^{f^l(t_1,\ldots,t_k)}, f^l(t_1, \ldots, t_k))\}$
 return Mem

As explained above, the collapse function returns the set Mem which we view as a substitution. Its application to a term or (more generally) to an assignation set L is written by $L\mathsf{Mem}$. We define Mem^{-1} to be $\{(u, g) \mid (g, u) \in \mathsf{Mem}\}$. It can be viewed as a replacement function. $L\mathsf{Mem}^{-1}$ denotes

L where each term u occurring as subterm in L such that $(u, g) \in \mathsf{Mem}^{-1}$ is replaced by g, where the replacement is applied top-most.

Function $\mathsf{saturate}^{\mathsf{tran}}(L)$
Let $\mathsf{Mem} = \mathsf{collapse}_{\Sigma_1,\Sigma_{\mathsf{tran}}}(L)$;
  (Mem links terms headed by transp. functions to garbage)
let $L' = L$
repeat
  let $L = L'$;
  let $L' = \mathsf{saturate}(L'\mathsf{Mem}^{-1})\mathsf{Mem}$;
    (L' is parsed according to the signature Σ_1)
    if there is $(b, g^l) \in L$ s.t. g^l is of sort term
    and $(M\mathsf{type})(b) \neq \bot$
      let $f = (M\mathsf{type})(b)$, f is of arity $s_1 \times \cdots \times s_k \to s$
    if there is $1 \leq i \leq k$ s.t. $(M\ \mathsf{proj}\ f\ i)(b) = \bot$
      then $L' = \overline{L'}^{g^l \mapsto g^l_s}$
      else let $b_i = (M\ \mathsf{proj}\ f\ i)(b)$, $1 \leq i \leq k$
      let $V_i = U_i$ if there exists $(b_i, U_i) \in L$,
        otherwise $V_i = [g^{l(b_i)}]$ with $l(b_i) \in \mathsf{labelsA}$;
      let $L' = L' \setminus \{(b, g^l)\} \cup$
       $\{(b, [f^l(g^{l(b_1)}, \ldots, g^{l(b_k)}); g^l]), (b_1, V_1), \ldots, (b_k, V_k)\}$
      (We update L' with the bitstring obtained
       by projection.)
      let $\mathsf{Mem} = \mathsf{Mem} \cup \{(g^l, f^l(g^{l(b_1)}, \ldots, g^{l(b_k)}))\}$
until $L = L'$
return L.

Above, $L^{u \mapsto v}$ is obtained from L by replacing each element $(c, [u])$ of L with $(c, [u, v])$

Figure 2: Extending parsing to transparent functions.

3. DEDUCTION SOUNDNESS

In this section we give a notion of soundness of an implementation with respect to a symbolic deduction relation. The definition is through a game which, informally, is as follows. The game, maintains internally an assignation set (that is, the interpretation of bitstrings as terms). The adversary is allowed two types of actions. First, he can see interpretation of whatever terms it wants: here the game runs internally the generate function defined earlier using the current assignation set and returns the bitstring that is obtained this way. Secondly, the adversary is allowed to see parsings of whatever bitstrings it wants: here the game runs internally the parse algorithm and returns the term that it obtains. Soundness holds if with overwhelming probability the bitstrings that the adversary asks to be parsed correspond to terms that can be deduced by the adversary from the terms for which it had seen interpretations. This definitional idea can be traced back to the work of Herzog [15] and captures the idea that a computational adversary (that operates with bitstrings) is essentially limited to performing Dolev-Yao operations on the bit-strings that it receives.

The novel aspect of our definition is that the implementation of primitives is not analyzed standalone. Instead, we consider an extended signature that includes a countable number of transparent functions. The implementation for these functions is selected by the adversary. In effect, our

Game $\mathsf{Game}_{\mathcal{S},\mathcal{F},A,M_1,\mathsf{parse},\mathcal{D}}(\eta)$:
Set $S = \emptyset$; (set of requested terms)
Set $L = \emptyset$; (assignation set)
$\mathsf{DY} = \top$; (only deducible terms have been received)
$\mathsf{Valid} = \top$; (only valid requests have been received)
$\mathcal{R} \leftarrow \{0,1\}^*$
$M_2 \leftarrow \mathcal{A}(\eta)$; where M_2 has to be a transparent
    implementation for $(\mathcal{S}, \mathcal{F} \setminus \mathcal{F}_1)$
On request parse c do:
  Compute $(t, L') := \mathsf{parse}^{\mathsf{tran}}(c, L)$;
  Let $L := L'$;
  If $S \not\vdash_{\mathcal{D} \cup \mathcal{D}_{\mathsf{tran}}(\mathcal{S},\mathcal{F}\setminus\mathcal{F}_1)} t$
    then $\mathsf{DY} = \bot$, output $(\mathsf{DY}, \mathsf{Valid})$, stop
    else return t
On request evaluate t do
  $S := S \cup \{t\}$;
  If $\neg P_{\mathsf{valid}}(S)$
    then $\mathsf{Valid} = \bot$, output $(\mathsf{DY}, \mathsf{Valid})$, stop
    else compute $(c, L') := \mathsf{generate}_{M_1 \cup M_2, \mathcal{R}}(t, L)$
    Let $L := L'$;
    return c
On request stop do
  output $(\mathsf{DY}, \mathsf{Valid})$, stop

Figure 3: Game defining deduction soundness.

game demands that the axiomatization of the primitives remains sound, even if the primitive is used in conjunction with arbitrary (adversarially chosen) other functions. This aspect of the definition is what enables compositionality of our notion.

Let $\Sigma = (\mathcal{S}, \mathcal{F})$ be a signature, a (symbolic) deduction system $\mathcal{D}$ associated to Σ and a concrete implementation M_1 for the functions $\mathcal{F}_1 \subset \mathcal{F}$. We define a game that captures the soundness of M_1 with respect to the deduction relation $\mathcal{D}$ in Figure 3. The game is parametrized by an adversary $\mathcal{A}$, the set of sorts $\mathcal{S}$ and a set of symbols $\mathcal{F}$ as above, a *$(\mathcal{S}, \mathcal{F}_1)$-concrete implementation* M_1 and, importantly, a predicate P_{valid} on a set of terms. This predicate restricts the queries of the adversary in various ways. For example, P_{valid} can be used to specify that the adversary is not allowed to create key cycles, only ask for well-formed terms, etc.. The last parameter is a parsing function parse. The game also maintains two flags DY and Valid initially set to true. The game allows three kinds of requests:

- The adversary sends a bit-string and gets a term that should represent the bit-string;
- The adversary sends a term t and gets its computational interpretation. In producing these interpretations the game needs random coins. These coins are generated as needed, so strictly speaking they are uniformly distributed in a finite set that depends on the running time of the adversary $\mathcal{A}$. To avoid introducing explicit bounds for this set we abuse notation and write $\mathcal{R} \leftarrow \{0,1\}^*$ for the (online) process of generating these coins.
- Finally, the adversary may decide to stop the game.

Two events may occur during the game:

- The flag DY is set to false if the adversary sends a request parse c such that the returned term t is non deducible symbolically, that is $S \not\vdash t$. This corresponds intuitively to cases where the adversary deviates from symbolic Dolev-Yao behaviors.
- The flag Valid is set to false if the list of requests S does not satisfy the predicate P_{valid} (which, as explained above, ensures that the adversary's requests are benign.)

We say that a computational implementation is *deduction sound* if the deduction relation $\vdash$ reflects the power of a computational adversary, that is if any bit-string sent by the adversary can be associated to a term deducible from previously seen terms.

DEFINITION 2. *Let $\mathcal{S}$ be a set of sorts and let $\mathcal{F}$ be a set of symbols. A $(\mathcal{S}, \mathcal{F}_1)$-concrete implementation M_1 is* deduction sound *w.r.t. a deduction system $\mathcal{D}$, a parsing function* parse *and a predicate P_{valid} if for any probabilistic polynomial time Turing machine (p.p.t) $\mathcal{A}$ (for which the game always stops)*

$$\mathbb{P}\left[\mathsf{Game}_{\mathcal{S},\mathcal{F},A,M_1,\mathsf{parse},\mathcal{D}}(\eta) = (\bot, \top)\right]$$

is a negligible *function of η, that is, remains eventually smaller than any η^{-n} $(n > 0)$ as η tends to infinity.*

Proving soundness in the sense of the definition above is not really more complex than when using other definitions in the literature. As an example, we prove the soundness of the implementation that we gave in our running example.

3.1 Soundness for signatures and MACs

In this section we show that the implementation of signatures and MACs specified in Section 2.2.4 is deduction sound. The predicate P_{sm} that restricts the queries of the adversary in the deduction soundness game only checks that signing keys have been generated before any signature generation request is issued (i.e. before the adversary sees the interpretation of a term that contains a signature).

PROPOSITION 1. *Let $\mathcal{F}_{\mathsf{tran}}$ be an arbitrary set of functional symbols with arity of the form* $\mathsf{term} \times \ldots \times \mathsf{term} \rightarrow \mathsf{term}$. *Assume that the digital signature scheme and MAC scheme used to define M_{sm} are* EU-CMA *secure. Then M_{sm} is a $(\mathcal{S}, \mathcal{F}_{\mathsf{sm}} \cup \mathcal{F}_{\mathsf{tran}})$-concrete implementation that is deduction sound with respect to $\mathcal{D}_{\mathsf{sm}}$, parsing function $\mathsf{parse}_{\mathsf{sm}}$ and predicate P_{sm}.*

Assume that there exists an adversary $\mathcal{A}$ which wins the game for deduction soundness with non-negligible probability. A simple structural induction shows that the term $f(t_1, t_2, \ldots, t_k)$ that is non Dolev-Yao (nonDY) according to the deduction relation induced by $\mathcal{D}_{\mathsf{sm}} \cup \mathcal{D}_{\mathsf{tran}}$ must contain a subterm which has its head symbol in $\mathcal{F}_{\mathsf{sm}}$. The only non-deducible terms in $\mathcal{F}_{\mathsf{sm}}$ are secret keys (signing or mac keys), and signatures or macs on messages that have not been signed (or mac'ed) before. Intuitively, this corresponds to the adversary breaking the primitives underlying the implementation M_{sm}. We formalize this intuition by constructing adversaries against the primitives, one against the signature scheme, and one against the mac scheme.

Here, we only explain how the adversary against the signature scheme works – the other adversary works analogously. Assume that we know which of the messages of the adversary $\mathcal{A}$ contains the nonDY subterm, and that this subterm is a signature valid under the public key of some party a on a message that this party did not produce. Also assume that we know the identity a of this party. Both of these quantities can be guessed with non-negligible probability. Adversary $\mathcal{B}$ that we construct against the digital signature receives as input some verification key vk and access to a signing oracle under the corresponding secret key sk. Adversary $\mathcal{B}$ then simulates the deduction soundness game for $\mathcal{A}$ in such a way that the signing key of party a, that is, the interpretation of $\mathsf{vk}(a)$ is set to vk. All of the other secrets in the game (mac keys, signing keys) are generated by $\mathcal{B}$. Adversary $\mathcal{B}$ simulates the behaviour of the game (that is algorithms $\mathsf{parse}^{\mathsf{tran}}$ and generate) as if on the internal assignation list where the verification key of a is set to vk. Whenever $\mathcal{B}$ needs to produce signatures under vk it uses its signing oracle. Notice that the adversary $\mathcal{A}$ never asks for the interpretation of $\mathsf{sk}(a)$ (which $\mathcal{B}$ cannot provide), as otherwise the signature would actually be a Dolev-Yao message. The nonDY message that $\mathcal{A}$ produces contains a signature that verifies under vk on some message m that a did not previously sign (and which thus was not queried to the signing oracle of $\mathcal{B}$). The same simulation can be provided by $\mathcal{B}$ when the nonDY message is the signing key associated to a. In this case, $\mathcal{B}$ simply signs some fresh message to produce its forgery.

4. COMPOSITION THEOREMS

Our notion of deduction soundness enjoys the nice property of being easily extendable: if a computational algebra is deduction sound for a given set of primitives, it is possible to add other primitives, one by one, without having to prove deduction soundness, from scratch for the resulting set of primitives. In this paper we provide two results: we show how to extend in a modular way deduction soundness to encompass public data structures and asymmetric encryption.

4.1 Adding public datastructures

An immediate observation with interesting implications is the following. Consider some implementation M_1 for the functions in $\mathcal{F}_1 \subseteq \mathcal{F}$ for some signature $(\mathcal{S}, \mathcal{F})$. Now, extend M_1 to implement, in a transparent way additional functions in $\mathcal{F} \setminus \mathcal{F}_1$. Then, the resulting implementation is also deduction sound. The intuition behind this result is simple: if M_1 is sound when the functions in $\mathcal{F} \setminus \mathcal{F}_1$ are implemented via a transparent implementation selected by the adversary, implementing some (or even all) of the functions in $\mathcal{F} \setminus \mathcal{F}_1$ with some fixed transparent implementation preserves soundness. This idea is formalized by the following theorem.

THEOREM 1. *Assume that M_1 is an $(\mathcal{S}, \mathcal{F}_1)$-concrete implementation for some signature $(\mathcal{S}, \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3)$ that is computationally sound with respect to deduction $\mathcal{D}$, parse function* parse *and predicate P_{valid}. Then, if M_2 is an arbitrary transparent implementation for the functions in $\mathcal{F}_2$, then $M_1 || M_2$ (the implementation that implements functions in $\mathcal{F}_1$ with M_1 and functions in $\mathcal{F}_2$ with M_2) is a $(\mathcal{S}, \mathcal{F}_1 \cup \mathcal{F}_2)$-concrete implementation for $(\mathcal{S}, \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3)$ that is deduction sound with respect to $\mathcal{D} \cup \mathcal{D}_{\mathsf{tran}}(\mathcal{F}_2)$, parse function $\mathsf{parse}^{\mathsf{tran}}$ and predicate P_{valid}. The function $\mathsf{parse}^{\mathsf{tran}}$ is defined via $\mathsf{saturate}^{\mathsf{tran}}$ (Figure 2) with the transparent implementation set to M_2.*

The proof of the theorem is by reduction. Consider an adversary $\mathcal{A}$ that breaks deduction soundness for the $(\mathcal{S}, \mathcal{F}_1 \cup \mathcal{F}_2)$-concrete implementation. We construct the following adversary $\mathcal{B}$ against the soundness of the implementation M_1. Adversary $\mathcal{B}$ runs internally $\mathcal{A}$: when $\mathcal{A}$ outputs its transparent interpretation M_3 for the functions in $\mathcal{F}_3$, adversary $\mathcal{B}$ outputs $M_2||M_3$ as its transparent interpretation for the functions in $\mathcal{F}_2 \cup \mathcal{F}_3$. After this, $\mathcal{B}$ simply forwards the queries of $\mathcal{A}$ to the game and forwards the returns. When $\mathcal{A}$ requests the parse of a string that corresponds to a nonDY term with respect to $\mathcal{D} \cup \mathcal{D}_{\mathsf{tran}(\mathcal{F}_2)}$ this term is in fact nonDY with respect to $\mathcal{D}$, so $\mathcal{B}$ wins its own game.

The above theorem is important as it allows to add, essentially for free, public data structures to any implementation that is deduction sound (in the sense of our definition) as typically data structures are implemented by transparent implementations.

For example lists can be added to a deduction soundness results as follows. Assume that the set of transparent functions $\mathcal{F}_{\mathsf{list}}$ contains two functions empty and @ with arities $\mathsf{empty} :\rightarrow \mathsf{term}$ and $@ : \mathsf{term} \rightarrow \mathsf{term}$, respectively. Consider the following implementation M_{list} for lists.

- $(M_{\mathsf{list}}\ \mathsf{evaluate}\ \mathsf{empty})$ outputs $\langle \epsilon, \mathsf{list} \rangle$ where ϵ is some fixed bitstring.
- $(M_{\mathsf{list}}\ \mathsf{evaluate}\ @)$ on input x_0 and U, parses U as $\langle x_1, x_2, \ldots, x_k, \mathsf{list} \rangle$ for some k. If $k = 1$ and x_1 is ϵ then return $\langle x_0, \mathsf{list} \rangle$, otherwise return $\langle x_0, x_1, \ldots, x_k, \mathsf{list} \rangle$.
- $(M_{\mathsf{list}}\ \mathsf{type})$ on input x parses U as $\langle x_1, \ldots, x_k, \mathsf{list} \rangle$ for some $k \geq 0$ and $x_1, x_2, \ldots, x_k$ arbitrary bitstrings. If parsing fails, then the algorithm returns $\perp$. If $k = 1$ and $x_1 = \epsilon$ then return empty, otherwise return @.
- $(M_{\mathsf{list}}\ \mathsf{proj}\ @\ 1)$ on input x attempts to parse x as $\langle x_1, x_2, \ldots, x_k, \mathsf{list} \rangle$. If this does not succeed it outputs $\perp$. Otherwise it outputs x_1.
- $(M_{\mathsf{list}}\ \mathsf{proj}\ @\ 2)$ on input x attempts to parse x as $\langle x_1, x_2, \ldots, x_k, \mathsf{list} \rangle$. If this does not succeed it outputs $\perp$. If $k = 1$ then return $\langle \epsilon, \mathsf{list} \rangle$, otherwise output $\langle x_2, x_3, \ldots, x_k, \mathsf{list} \rangle$.

The above implementation clearly satisfies the properties of a transparent implementation. Theorem 1 works for arbitrary sets of transparent symbols. By replacing in the statement of the theorem $\mathcal{F}_2$ with $\mathcal{F}_{\mathsf{list}}$, and M_2 with $\mathcal{M}_{\mathsf{list}}$ one obtains a soundness result for the extension of the implementation of M_1 with M_{list} with respect to the deducibility relation induced by $\mathcal{D}_{\mathsf{sm}} \cup \mathcal{D}_{\mathsf{list}}$. Here, D_{list} is simply $\mathcal{D}_{\mathsf{tran}}(\mathcal{F}_{\mathsf{list}})$. We use this notation later in the paper.

Of course, other possible implementations for lists are possible and the theorem can be reused for each of these different implementations (as long as they are transparent). It is also possible to provide similar specifications for the usual pairing operation or more complex data structures (e.g. XML documents), and the corresponding extension result holds.

4.2 Adding asymmetric encryption

In this section we give a theorem that allows to extend a deduction sound implementation with asymmetric encryption. Our modeling assumes that corrupted keys are "honestly" generated by the key generation for the primitive. This assumption is quite common in existing soundness results (e.g.[5, 13, 16]) and is rarely circumvented (e.g.[8, 18]). For asymmetric primitives, as in the case of asymmetric encryption, the assumption can be ensured through a PKI. The reason for this assumption is that the problems outlined in [10] for the case of symmetric encryption also apply for asymmetric encryption.

For an algebraic signature $\mathcal{S}, \mathcal{F}$ we define its extension $\mathcal{S}_{\mathsf{enc}}, \mathcal{F}_{\mathsf{enc}}$ to asymmetric encryption. We define

$$\mathcal{S}_{\mathsf{enc}} = \mathcal{S} \cup \{\mathsf{id}, \mathsf{keypair}, \mathsf{ekey}, \mathsf{dkey}\}$$

where id represents agent identities and sorts $\mathsf{keypair}$, ekey, dkey contain key pairs, keys for encryption, and decryption respectively. We assume $\mathsf{keypair}, \mathsf{ekey}, \mathsf{dkey} \notin \mathcal{S}$ while id may already be in $\mathcal{S}$. Moreover, id is a sub-sort of term. We consider $\mathcal{F}_{\mathsf{enc}} = \mathcal{F} \uplus \{\mathsf{keypair}, \mathsf{ek}, \mathsf{dk}, \mathsf{enc}\}$. The function $\mathsf{keypair}$ of arity $\mathsf{id} \rightarrow \mathsf{keypair}$ takes on input an identity and returns the key pair associated to the identity. The function ek of arity $\mathsf{keypair} \rightarrow \mathsf{ekey}$ (resp dk of arity $\mathsf{keypair} \rightarrow \mathsf{dkey}$) is defined on the sort $\mathsf{keypair}$ and returns the encryption key (resp. the decryption key) associated to the key pair. The encryption function enc has arity $\mathsf{ekey} \times \mathsf{term} \rightarrow \mathsf{ciphertext}$. Note that, by construction, well-sorted terms do not contain strict subterms of sort dkey. By abuse of notation, we will often write $\mathsf{ek}(a)$ instead of $\mathsf{ek}(\mathsf{keypair}(a))$.

Let $\vdash$ be a deduction relation defined by a deduction system $\mathcal{D}$. We extend this deduction system with rules that capture the security of encryption:

$$\mathcal{D}_{\mathsf{enc}} = \left\{ \begin{array}{c} \frac{\mathsf{enc}^l(\mathsf{ek}(v),u) \quad \mathsf{dk}(v)}{u}, \qquad \frac{\mathsf{ek}(v) \quad u}{\mathsf{enc}^l(\mathsf{ek}(v),u)} l \in \mathsf{labelsA}, \\ \\ \frac{\mathsf{enc}^l(\mathsf{ek}(v),u)}{u} l \in \mathsf{labelsA} \end{array} \right\}$$

where v is a variable of sort id and u is a variable of sort term. We write $\vdash_{\mathsf{enc}}$ for the deduction relation induced by $\mathcal{D} \cup \mathcal{D}_{\mathsf{enc}}$

We now give a concrete implementation M_{enc} for encryption. The implementation uses some asymmetric encryption scheme $\Pi = (\mathcal{G}, \mathcal{E}, \mathcal{D})$. As usual, here $\mathcal{G}$ is a generation algorithm for key pairs, $\mathcal{E}$ is an encryption algorithm and $\mathcal{D}$ is a decryption algorithm. Note that $\mathcal{E}$ is an algorithm that takes three inputs: the encryption key, the message to be encrypted and the randomness that is used for encryption. We consider an implementation that affixes to bitstring a tag that indicates how it has been generated. The four possible tags are $\mathsf{tenc}, \mathsf{tkeypair}, \mathsf{tek}$ and tdk for cyphertexts, keypairs, encryption keys and decryption keys resp. The computable interpretations of $\mathsf{keypair}, \mathsf{ek}, \mathsf{dk}, \mathsf{enc}$ are as follows:

- $(M_{\mathsf{enc}}\ \mathsf{keypair}\ a)$ is $\mathcal{G}(\eta)$ to the output of which we append the tag $\mathsf{tkeypair}$;
- $(M_{\mathsf{enc}}\ \mathsf{enc}) : (k, m, r)$: parse k as $\langle pk, \mathsf{tek} \rangle$, compute $c \leftarrow \mathcal{E}(pk, m, r)$ and output $\langle c, pk, \mathsf{enc} \rangle$.
- $(M_{\mathsf{enc}}\ \mathsf{ek})$ is the function that extracts the encryption key from a key pair and adds the tag tek;
- $(M_{\mathsf{enc}}\ \mathsf{dk})$ is the function that extracts the decryption key from a key pair and adds the tag tdk.

Next, we show how to extend the parsing parse algorithm for the original implementation M to take into account the newly added primitive. In Figure 4 we define the function $\mathsf{saturate}^{\mathsf{enc}}$ (that in turn defines the parsing algorithm

$\mathsf{parse}^{\mathsf{enc}}$), starting from the abstract algorithm $\mathsf{saturate}$ underlying the parsing algorithm for the implementation M. The algorithm works as expected: it tries to interpret the strings that are parsed as garbage symbols by the original algorithm (i.e. $\mathsf{saturate}$) as encryptions. If this does not succeed, the interpretation of these bitstrings does not change. Otherwise, the bitstrings are decrypted and the plaintext is then parsed according to $\mathsf{saturate}$. The process is repeated until the assignation set remains unchanged (i.e. it is not possible to parse some bitstring that is interpreted as garbage in the current assignation set).

We assume that whenever there is $(ek, \mathsf{ek}(a)) \in L$ then there is (a unique) $(dk, \mathsf{dk}(a)) \in L$. It is easy to check that this property is preserved by the $\mathsf{saturate}^{\mathsf{enc}}$ function.

Function $\mathsf{saturate}^{\mathsf{enc}}(L)$
Let $\mathsf{Mem} = \mathsf{collapse}_{\Sigma,\Sigma_{\mathsf{enc}}}(L)$;
(Mem links terms headed by functions outside Σ to garbage)
let $L' = L$
repeat
let $L = L'$;
let $L' = \mathsf{saturate}(L'\mathsf{Mem}^{-1})\mathsf{Mem}$;
(L' is parsed according to the signature Σ)
if there is $(b, g^l) \in L$ where g is of sort term
and b is of the form $\langle b', ek, \mathsf{enc}\rangle$,
and there is $(ek, \mathsf{ek}(a)) \in L$ thus a unique $(dk, \mathsf{dk}(a)) \in L$,
and $\mathcal{D}(b', dk) \neq \bot$, do
let $m = \mathcal{D}(b', dk)$,
if there is $(m, U) \in L'$ then let $V = U$
otherwise let $V = [g^{l(m)}]$ where $l(m) \in \mathsf{labelsA}$;
let $L' = L' \setminus \{(b, g^l)\} \cup \{(b, [\mathsf{enc}^l(\mathsf{ek}(a), g^{l(m)}); g^l]), (m, V)\}$
let $\mathsf{Mem} = \mathsf{Mem} \cup \{(g^l, \mathsf{enc}^l(\mathsf{ek}(a), g^{l(m)}))\}$
until $L = L'$
return L

Figure 4: Extending parsing to asymmetric encryption.

We now show that the extension of a deduction-sound implementation with asymmetric encryption, as specified above remains deduction-sound, provided that the encryption scheme used in the implementation is $\mathsf{IND\text{-}CCA}$ secure. More precisely, we show that soundness of the extension holds with respect to deduction system $\mathcal{D} \cup \mathcal{D}_{\mathsf{enc}}$ and parsing function $\mathsf{parse}^{\mathsf{enc}}$. The predicate that validates the requests of the adversary needs to be changed (as the signature has changed). In particular, the generate requests in the new game should satisfy that requests remain valid w.r.t. to the initial signature and some additional constraints due to the use of encryption. Formally, given a predicate P_{valid} for a signature $(\mathcal{S}, \mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}})$, $\overline{P_{\mathsf{valid}}}^{\mathsf{enc}}$ is any predicate that conforms to the following restrictions (one would consider the least restrictive version).

- The adversary may generate keys (i.e. perform requests $\mathsf{generate\ ek}(\mathsf{keypair}(a))$ or $\mathsf{generate\ keypair}(a)$, with $a \in \mathsf{id}$) but only at the beginning of the game (before any encryption request is made); this requirement corresponds to allowing only static corruption of encryption keys.

- After this first phase, $\mathsf{keypair}(a)$ do not do not appear in terms except inside a sub-term of the form $\mathsf{ek}(\mathsf{keypair}(a))$;

- Importantly, the requests for the implementation that includes encryption, should remain valid w.r.t. the initial signature. To specify an appropriate extension of P_{valid} we remark the following about the workings of the game. As far as the adversary is concerned, ciphertexts are either entirely opaque (encrypted under honest keys) or entirely transparent (encrypted under corrupted keys). We can interpret then bitstrings with respect to the original signature as either garbage constants or as transparent functions. The validity requirement is then that terms with respect to the joint signature are, under this substitution valid with respect to the game for the original signature. Formally, we capture this idea as follows.

  We define the auxiliary function $\mathsf{replace}_{(H,C)}$ which is parameterized by two sets of agents H and C such that $\mathsf{id} = H \cup C$, from terms in $\mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}} \cup \mathcal{F}_{\mathsf{enc}}$ to terms in $\mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}}$ as follows:

  - $\mathsf{replace}_{(H,C)}(f^l(t_1, \ldots, t_n)) = f^l(\mathsf{replace}_{(H,C)}(t_1), \ldots, \mathsf{replace}_{(H,C)}(t_n))$ if $f \neq \mathsf{enc}$;
  - $\mathsf{replace}_{(H,C)}(\mathsf{enc}^l(\mathsf{ek}(b), t)) = f^l_{\mathsf{ek}(b)}(\mathsf{replace}_{(H,C)}(t))$ if $b \in C$;
  - $\mathsf{replace}_{(H,C)}(\mathsf{enc}^l(\mathsf{ek}(a), t)) = f^{l(\mathsf{enc}^l(\mathsf{ek}(a),t))}_{\mathsf{ek}(a)}$ if $a \in H$ and $l \in \mathsf{labelsH}$;
  - $\mathsf{replace}_{(H,C)}(\mathsf{enc}^l(\mathsf{ek}(a), t)) = g^{\mathsf{enc}^l(\mathsf{ek}(a),t)}$ if $a \in H$ and $l \in \mathsf{labelsA}$;

  Then, $\overline{P_{\mathsf{valid}}}$ is any predicate such that for any choice of H and C such that $\mathsf{id} = H \cup C$ the following holds:

$$\mathsf{replace}_{(H,C)}(\overline{P_{\mathsf{valid}}}^{\mathsf{enc}}) \subseteq P_{\mathsf{valid}} \qquad (1)$$

where $f^l_{\mathsf{ek}(b)}$ is a unary symbol and $f^{l(\mathsf{enc}^l(\mathsf{ek}(a),t))}_{\mathsf{ek}(a)}$ is a constant symbol.

Our next theorem shows that any deduction sound set of primitives can be automatically extended to asymmetric encryption.

THEOREM 2. *Let $\mathcal{S}_1$ be a set of sorts, $\mathcal{F}_1$ be a set of symbols. Let $\mathcal{F}_{\mathsf{tran}}$ and $\mathcal{F}_{\mathsf{tran}'}$ be two disjoint sets of functions symbols of arity of the form $\mathsf{term} \times \cdots \times \mathsf{term} \to \mathsf{term}$ such that they contain an infinite countable number of functions of each arity. Let M_1 be a $(\mathcal{S}_1, \mathcal{F}_1 \cup \mathcal{F}_{\mathsf{tran}} \cup \mathcal{F}_{\mathsf{tran}'})$-concrete implementation which is deduction sound w.r.t. a deduction system $\mathcal{D}$, a parsing function parse and a predicate P_{valid}. Let $\mathcal{S}_{\mathsf{enc}}$, $\mathcal{F}_{\mathsf{enc}}$, $\mathcal{D}_{\mathsf{enc}}$, M_{enc} as defined in Section 4.2.*

Then $(M_1 \| M_{\mathsf{enc}})$ is a deduction sound $((\mathcal{S}_1 \cup \mathcal{S}_{\mathsf{enc}}, \mathcal{F}_1 \cup \mathcal{F}_{\mathsf{enc}} \cup F_{\mathsf{tran}}))$-concrete implementation w.r.t. the deduction system $\mathcal{D} \cup \mathcal{D}_{\mathsf{enc}}$, the parsing function $\mathsf{parse}^{\mathsf{enc}}$ and any predicate $\overline{P_{\mathsf{valid}}}^{\mathsf{enc}}$ that satisfies Equation 1, provided that the encryption scheme used by M_{enc} is $\mathsf{IND\text{-}CCA}$.

The proof of Theorem 2 works in two stages. First, we transform the deduction-soundness game for the $(\mathcal{S}_1 \cup \mathcal{S}_{\mathsf{enc}}, \mathcal{F}_1 \cup \mathcal{F}_{\mathsf{enc}} \cup \mathcal{F}_{\mathsf{tran}})$-concrete implementation $(M_1 \| M_{\mathsf{enc}})$ into a game $\mathsf{Game}^{\mathsf{fake}}$ where the encryption function is replaced by a function that encrypts with zeros when called on "honest" identities. The sets H and D of honest and dishonest identities

are determined during the first phase of the game, when the adversary does not request any encryption: dishonest identities are those for which the adversary knows both the public and private corresponding keys while honest identities are the remaining ones. The IND-CCA assumption on the encryption function ensures that the two games are indistinguishable.

The second (and main) step of the proof is a reduction. Assume the existence of an adversary $\mathcal{A}$ that breaks the game $\mathsf{Game}^{\mathsf{fake}}$. Building upon $\mathcal{A}$, we construct an adversary $\mathcal{B}$ that wins the game $\mathsf{Game}_{\mathcal{S}_1,\mathcal{F}_1\cup\mathcal{F}_{\mathsf{tran}}\cup\mathcal{F}_{\mathsf{tran}'},\mathcal{B},M_1,\mathsf{parse},\mathcal{D}}$, that is, an adversary that contradicts the soundness of implementation M_1. In the first phase, $\mathcal{B}$ generates the key pairs (honest and dishonest) itself. The key idea is to show how the transparent functions in $\mathcal{F}_{\mathsf{tran}'}$ can be used by $\mathcal{B}$ to apply encryption inside terms it does not have access to. The simulation of encryption by transparent function works as follows. For each dishonest key $\mathsf{ek}(b) \in D$, we assume given a transparent function $f_{\mathsf{ek}(b)} \in \mathcal{F}_{\mathsf{tran}'}$ of arity 1, whose interpretation (provided by $\mathcal{B}$) is to encrypt with $\mathsf{ek}(b)$. The associated projector is simply the decryption with the corresponding decryption key. For each honest key $\mathsf{ek}(a)$ and length $l \in \mathbb{N}$, we assume given a transparent function $f_{\mathsf{ek}(a),l} \in \mathcal{F}_{\mathsf{tran}'}$ of arity 0 (a constant) that simply encrypts 0^l by $\mathsf{ek}(a)$. This function symbol being a constant, it does not have a projector. We can then conclude the proof by showing that $\mathcal{B}$ interacting with $\mathsf{Game}_{\mathcal{S}_1,\mathcal{F}_1\cup\mathcal{F}_{\mathsf{tran}}\cup\mathcal{F}_{\mathsf{tran}'},\mathcal{B},M_1,\mathsf{parse},\mathcal{D}}$ exactly simulates $\mathsf{Game}^{\mathsf{fake}}$.

5. APPLICATIONS

In this section we present two applications of our notion. First, we show that the composability property of our soundness notion allows to combine, several different primitives: we obtain deduction soundness for mac, signatures, lists and asymmetric encryption. In the following proposition $\mathsf{parse}_{\mathsf{sm},\mathsf{list},\mathsf{sm}}$ is the parsing algorithm $\mathsf{parse}_{\mathsf{sm}}$ extended to lists and encryption as specified earlier in the paper and predicate $\overline{P_{\mathsf{sm}}}$ is any predicate that satisfies Equation 1 (where P_{valid} is replaced by P_{sm}) and $\mathsf{parse}_{\mathsf{sm},\mathsf{list},\mathsf{sm}}$.

PROPOSITION 2. *$M_{\mathsf{sm}}||M_{\mathsf{list}}||M_{\mathsf{enc}}$ is a concrete implementation for $(\mathcal{S}_{\mathsf{enc}}, (\mathcal{F}_{\mathsf{sm}} \cup \mathcal{F}_{\mathsf{list}})_{\mathsf{enc}})$ that is deduction sound with respect to $\mathcal{D}_{\mathsf{sm}}\cup\mathcal{D}_{\mathsf{list}}\cup\mathcal{D}_{\mathsf{enc}}$, parsing function $\mathsf{parse}_{\mathsf{sm},\mathsf{list},\mathsf{enc}}$ and predicate $\overline{P_{\mathsf{sm}}}$, if the digital signature scheme and the MAC scheme used in M_{sm} are* EU-CMA *secure and the encryption scheme used in M_{enc} is* IND-CCA *secure.*

The proof of the proposition is an immediate consequence of Proposition 1 and Theorems 1 and 2.

Next, we show that deduction soundness allows computationally sound symbolic analysis for a wide class of protocols. We specify this class through an abstract property that links symbolic and computational versions of the same protocol. To this end, we regard protocols as state transition systems. So, a *symbolic protocol* is simply a state-based transition system that takes terms as inputs and outputs terms. A *computational protocol* is state-based transition system that takes bitstrings as inputs and outputs bitstrings. Symbolic protocols could be specified for instance using some process calculus, whereas computational protocols are specified through communicating Turing machines.

The property that we define next, and which we call *commutation property* links symbolic and computational protocols. Intuitively the property says that the computational interpretation of the symbolic behavior of the protocol correspond to its implementation.

DEFINITION 3 (COMMUTATION PROPERTY). *Let P^s be a symbolic protocol and P^c be a computational protocol. Intuitively, P^s and P^c should be respectively the symbolic and computational interpretation of the same protocol P. Let $\mathcal{S}$ be a set of sorts, $\mathcal{F}$ be a set of symbols. Let M_1 be a $(\mathcal{S},\mathcal{F}_1)$-concrete implementation and* parse *be a parsing function. We say that (P^s, P^c)* has the commutation property *if the two following games are computationally indistinguishable by any polynomial adversary.*

1. *Game 1: $\mathcal{A}$ interacts with P^c directly.*
2. *Game 2:*

   Set $L = \emptyset$; (assignation set)
   On input c from $\mathcal{A}$ do:
   $(t, L) := \mathsf{parse}(c, L)$
   send t to P^s, getting u
   $(c', L) := \mathsf{generate}_{M_1}(u, L)$
   return c' to $\mathcal{A}$

For example, for any protocol described by the language of [13] (with input and output of messages, conditional with no else branch and replication of processes) the formal and computational interpretations of the protocol enjoy the commutation property. A similar property has been identified by Backes, Hofheinz, and Unruh as a sufficient condition to obtain soundness results in their CoSP framework [3]. They elaborate in the full version of their paper and show how to prove this commutation property for all protocols written in a specific language. Similar results are possible within the framework that we propose.

Next, we show that for protocols that satisfy this commutation property one can prove a soundness result: whenever a formal protocol enjoys a security property (for a symbolic adversary), the corresponding computational protocol also enjoys the same security property. The focus of this paper is on trace-based properties which are naturally specified in terms of deducibility relations.

We write $P \xrightarrow{?u!v} P'$ if a protocol at state P outputs v on input u, moving to state P'. A trace is then a sequence of input/output messages. Formally, a *formal* (resp. *computational*) *execution trace* of a formal (resp. computational) protocol P is a sequence of the form $?u_1!v_1?u_2!v_2\cdots?u_n!v_n$ such that the u_i, v_i are terms (resp. bitstrings) and $P \xrightarrow{?u_1!v_1} P_1?u_2!v_2 \longrightarrow P_2 \cdots ?u_n!v_n \longrightarrow P_n$. For formal protocols, we say that a formal execution trace $?u_1!v_1?u_2!v_2\cdots?u_n!v_n$ is *valid* w.r.t. to a deduction system $\mathcal{D}$ if $\{v_1, \ldots, v_{i-1} \vdash_{\mathcal{D}} u_i\}$ for any $1 \leq i \leq n$. Valid execution traces ensures that the symbolic adversary only makes valid (deducible) queries. A security property can be any predicate on the execution traces.

DEFINITION 4. *Let ϕ^s (resp. ϕ^c) be a predicate on formal (resp. computational) traces. Let P^s be a symbolic protocol and P^c be a computational protocol. Let $\mathcal{D}$ be a deduction system. We say that P^s satisfies ϕ^s, denoted $P^s \models^s \phi^s$ if ϕ^s holds for any valid execution trace of P^s, w.r.t. $\mathcal{D}$. Similarly, we say P^c satisfies ϕ^c, denoted $P^c \models^c \phi^c$ if for any p.p.t. adversary $\mathcal{A}$, the predicate ϕ^c holds with overwhelming probability on execution traces of P^c interacting with $\mathcal{A}$.*

Deduction soundness allows the transfer of security properties for any formal / computation interpretation that enjoys the commutation property from the symbolic to the computational setting.

PROPOSITION 3. *Let P^s be a symbolic protocol and P^c be a computational protocol. Let ϕ^s be a predicate on formal traces. Let $\mathcal{S}$ be a set of sorts, $\mathcal{F}$ a set of symbols, $\mathcal{D}$ a deduction system, P_{valid} a predicate and* parse *a parsing function. Assume M_1 is a deduction sound $(\mathcal{S}, \mathcal{F}_1)$-concrete implementation, w.r.t. $\mathcal{D}$, P_{valid} and* parse*. Assume that (P^s, P^c) has the commutation property and that all valid executions of P_s satisfy P_{valid}. Then*

$$P^s \models^s \phi^s \Rightarrow P^c \models^c \phi^c$$

where ϕ^c is the image of ϕ^s. Formally, $\phi^c = \{t^c \mid t^c = L(t^s)$ for some $t^s \in \phi^s, L \in \mathcal{L}\}$ where $\mathcal{L}$ is the set of all L that can be constructed in Game 2.

6. CONCLUSION

In this paper we propose *deduction soundness*, a novel computational soundness framework. The key feature of our notion is that it allows for modular extensions and we provide two concrete examples. We show that implementations for arbitrary cryptographic primitives that are deduction sound can be extended with public data structures and asymmetric encryption without repeating the original proof effort.

There are several obviously interesting directions for further work. The first is to show that deduction soundness can be extended, modularly, to other cryptographic primitives. We expect that the techniques developed in this paper for the case of asymmetric encryption would apply virtually unchanged to the case of hash functions (modeled as random oracles) and symmetric encryption schemes. However, in its present form, deduction soundness does not extend to authentication primitives (e.g. digital signature and message authentication codes). The main reason is that the current definition does not account for the interaction between the axioms that are proved deduction sound and those that characterize the implementation provided by the adversary, as far as authentication aspects are concerned. A modular extension with signatures and macs needs to limit, in a fairly non-restrictive manner, the interaction between the axioms for these authentication primitives and those of for which soundness had already been proved. We leave such an extension for further work.

The notion of deduction soundness is concerned directly with the implementation of cryptographic primitives and is independent of any particular protocol specification language. Computational soundness for such languages can however still be established if a certain property which we call a commutation property holds. The commutation property ensures that a real execution of a protocol (with primitives implemented as expected) is the same as the execution of the protocol with primitives implemented as in the game that defines soundness. If this property holds, then a real execution of the protocol can be mapped to a symbolic one and this, in turn, leads to a soundness result in the form of a mapping lemma.

Mapping lemmas allow for the translation of authentication properties from symbolic models to computational ones but they do not directly yield similar results for secrecy properties. An intriguing open question is therefore to develop modularly extensible frameworks for computational soundness that allow for the transfer of secrecy properties. Specification through equivalence notions (rather than deducibility) lend themselves naturally to capturing secrecy properties (both symbolically and computationally). A possible direction would therefore be to define *equivalence soundness*, the analogous of our notion for equivalences (rather than deducibility).

Finally, we note that proving that the commutation property holds for a general specification language is not particularly difficult, but it is rather tedious, as evidenced by the work of Backes, Hofheinz, and Unruh (in the long version of [3]), where such a property had been proved. An interesting research question is to find ways to prove, also in a modular way that the commutation property holds.

Acknowledgement.

We would like to thank Steve Kremer, Hubert Comon Lundh, Ralf Küsters, and Dominique Unruh for useful discussions and suggestions. In particular, Dominique suggested the search for a modular way of proving the commutation property. This work has been supported in part by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and by the European Research Council under the European Union's Seventh Framework Programme (FP72007–2013)/ERC grant agreement 258865 - project ProSecure.

7. REFERENCES

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[2] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. Abad. Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, pages 1–10, 2008.

[3] M. Backes, D. Hofheinz, and D. Unruh. Cosp: A general framework for computational soundness proofs. In *ACM CCS 2009*, pages 66–78, November 2009. Preprint on IACR ePrint 2009/080.

[4] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE Computer Science Foundations Workshop (CSFW'04)*, pages 204–218, 2004.

[5] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

[6] M. Backes and D. Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 255–269, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[7] R. Canetti. Universally composable signature, certification, and authentication. In *Proc. 17th IEEE*

Computer Security Foundations Workshop (CSFW'04), pages 219–235, 2004.

[8] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols (extended abstract). In *Proc. 3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 380–403. Springer, 2006.

[9] H. Comon-Lundh and V. Cortier. Computational soundness of observational equivalence. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, Virginia, USA, Oct. 2008. ACM Press.

[10] H. Comon-Lundh and V. Cortier. How to prove security of communication protocolsň A discussion on the soundness of formal models w.r.t. computational ones. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29–44, 2011.

[11] V. Cortier, S. Kremer, R. Küsters, and B. Warinschi. Computationally sound symbolic secrecy in the presence of hash functions. In *Proceedings of the 26th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *LNCS*, pages 176–187, Kolkata, India, 2006. Springer.

[12] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011.

[13] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 157–171, Edinburgh, UK, 2005. Springer.

[14] D. Galindo, F. D. Garcia, and P. van Rossum. Computational soundness of non-malleable commitments. In *Proc. 4th Information Security Practice and Experience Conference (ISPEC'08)*, LNCS, 2008. To appear.

[15] J. Herzog. A computational interpretation of Dolev-Yao adversaries. *Theoretical Computer Science*, 340:57–81, June 2005.

[16] R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 172–185. Springer, 2005.

[17] R. Janvier, Y. Lakhnech, and L. Mazaré. Computational soundness of symbolic analysis for protocols using hash functions. In *Proceedings of the Workshop on Information and Computer Security (ICS'06)*, Electronic Notes in Theoretical Computer Science, Timisoara, Romania, Sept. 2006. Elsevier Science Publishers.

[18] R. Küsters and M. Tuengerthal. Computational Soundness for Key Exchange Protocols with Symmetric Encryption. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 91–100. ACM Press, 2009.

[19] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, March 1996.

[20] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC'04)*, volume 2951 of *LNCS*, pages 133–151. Springer, 2004.