Computational Soundness without Protocol Restrictions

Michael Backes[1], Ankit Malik[2] and Dominique Unruh[3]
[1] Saarland University, Germany and MPI-SWS
[2] Dept. of Math., IIT Delhi
[3] University of Tartu, Estonia

ABSTRACT

The abstraction of cryptographic operations by term algebras, called Dolev-Yao models, is essential in almost all tool-supported methods for verifying security protocols. Recently significant progress was made in establishing computational soundness results: these results prove that Dolev-Yao style models can be sound with respect to actual cryptographic realizations and security definitions. However, these results came at the cost of imposing various constraints on the set of permitted security protocols: e.g., dishonestly generated keys must not be used, key cycles need to be avoided, and many more. In a nutshell, the cryptographic security definitions did not adequately capture these cases, but were considered carved in stone; in contrast, the symbolic abstractions were bent to reflect cryptographic features and idiosyncrasies, thereby requiring adaptations of existing verification tools.

In this paper, we pursue the opposite direction: we consider a symbolic abstraction for public-key encryption and identify two cryptographic definitions called PROG-KDM (programmable key-dependent message) security and MKE (malicious-key extractable) security that we jointly prove to be sufficient for obtaining computational soundness without imposing assumptions on the protocols using this abstraction. In particular, dishonestly generated keys obtained from the adversary can be sent, received, and used. The definitions can be met by existing cryptographic schemes in the random oracle model. This yields the first computational soundness result for trace-properties that holds for arbitrary protocols using this abstraction (in particular permitting to send and receive dishonestly generated keys), and that is accessible to all existing tools for reasoning about Dolev-Yao models without further adaptations.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol Verification*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

General Terms

Security, theory, verification

Keywords

Computational soundness, sending keys, key dependent messages

1. INTRODUCTION

Proofs of security protocols are known to be error-prone and, owing to the distributed-system aspects of multiple interleaved protocol runs, awkward for humans to make. Hence work towards the automation of such proofs started soon after the first protocols were developed. From the start, the actual cryptographic operations in such proofs were idealized into so-called Dolev-Yao models, following [23, 24, 30], e.g., see [25, 34, 1, 28, 33, 13]. This idealization simplifies proof construction by freeing proofs from cryptographic details such as computational restrictions, probabilistic behavior, and error probabilities. The success of these Dolev-Yao models for the tool-supported security analysis stems from their conceptual simplicity: they only consist of a small set of explicitly permitted rules that can be combined in an arbitrary manner, without any further constraints on the usage and combination of these rules. Recently significant progress was made in establishing so-called computational soundness results: these results prove that Dolev-Yao style models can be sound with respect to actual cryptographic realizations and security definitions, e.g., see [2, 26, 10, 8, 27, 31, 22, 19, 11, 21].

However, prior computational soundness results came at the price of imposing various constraints on the set of permitted protocols. In addition to minor extensions of symbolic models, such as reflecting length information or randomization, core limitations were to assume that the surrounding protocol does not cause any key cycles, or – more importantly – that all keys that are used within the protocol have been generated using the correct key generation algorithm. The latter assumption is particularly problematic since keys exchanged over the network might have been generated by the adversary, and assuming that the adversary is forced to honestly generate keys can hardly be justified in practice.

In a nutshell, these constraints arose because the respective cryptographic security definitions did not adequately capture these cases, but were considered carved in stone; in contrast, the symbolic abstractions were bent to reflect cryptographic features and idiosyncrasies. As a result, existing

tools needed to be adapted to incorporate extensions in the symbolic abstractions, and the explicitly imposed protocol constraints rendered large classes of protocols out-of-scope of prior soundness results. Moreover, if one intended to analyze a protocol that is comprised by such prior results, one additionally had to formally check that the protocol meets the respective protocol constraints for computational soundness, which is not necessarily doable in an automated manner.

Our Contribution. In this paper, we are first to pursue the opposite direction: we consider an unconstrained symbolic abstraction for public-key encryption and we strive for avoiding assumptions on the protocols using this abstraction. We in particular permit sending and receiving of potentially dishonestly generated secret keys. Being based on the CoSP framework, our result is limited to trace properties. We do not, however, see a principal reason why it should not be possible to extend it to equivalence properties.

To this end, we first identify which standard and which more sophisticated properties a cryptographic scheme for public-key encryption needs to fulfill in order to serve as a computationally sound implementation of an unrestricted Dolev-Yao model, i.e., eliminating constraints on the set of permitted cryptographic protocols. This process culminates in the novel definitions of an PROG-KDM (programmable key-dependent message) secure and an MKE (malicious-key extractable) secure encryption scheme. Our main result will then show that public-key encryption schemes that satisfy PROG-KDM and MKE security constitute computationally sound implementations of unrestricted Dolev-Yao models for public-key encryption. The definitions can be met by existing public-key encryption schemes. (A number of additional conditions are needed, e.g., that a public key can be extracted from a ciphertext. But these can be easily enforced by suitable tagging. See Appendix A for the full list.)

Our computational soundness result in particular encompasses protocols that allow honest users to send, receive and use dishonestly generated keys that they received from the adversary, without imposing further assumptions on the symbolic abstraction. This solves a thus far open problem in the cryptographic soundness literature.[1]

In a nutshell, we obtain the first computational soundness result that avoids to impose constraints on the protocols using this abstraction (in particular, it permits to send, receive, and use dishonestly generated keys), and that is accessible to all existing tools for reasoning about Dolev-Yao models without further adaptations.

Related work. Backes, Pfitzmann, and Scedrov [9] give a computational soundness result allowing key-dependent messages and sending of secret keys. But they impose the protocol condition that no key that is revealed to the adversary is every used for encrypting. Adão, Bana, Herzog, and Scedrov [3] give a computational soundness result allowing key-dependent messages, but only for passive adversaries. No adaptive revealing of secret keys is supported. Mazaré and Warinschi [29] give a computational soundness that allows for adaptive revealing of secret keys (in the case of symmetric encryption). But they disallow key-dependent messages, encrypting of keys, key-dependent messages, encryptions of ciphertexts, or forwarding of ciphertexts. They show that under these conditions, IND-CCA2 security is sufficient. Bana and Comon-Lundh [12] have a computational soundness result not imposing any restrictions on the protocol. Their symbolic modeling, however, is weakened so that no secrecy (even symbolically) is guaranteed when key dependent messages or adaptive revealing of secret keys occur.

Outline of the Paper. First, we introduce our symbolic abstraction of unconstrained public-key encryption within the CoSP framework in Section 2. We give the notion of computation soundness in Section 3 and review how prior computational soundness proofs were conducted in CoSP in Section 4 for the sake of illustration. We identify where the aforementioned restrictions arise in these proofs and explain how to overcome these limitations in Section 5. The corresponding formal result is established in Section 6. Full proofs are deferred to the full version [7]

2. THE SYMBOLIC MODEL

We first describe our symbolic modeling here. The model is fairly standard and follows that of [4], except that we added some additional operations on secret keys.

Constructors and nonces. Let $\mathbf{C} := \{enc/3, ek/1, dk/1, sig/3, vk/1, sk/1, pair/2, string_0/1, string_1/1, empty/0, garbageSig/2, garbage/1, garbageEnc/2\}$ and $\mathbf{N} := \mathbf{N}_P \cup \mathbf{N}_E$. Here $\mathbf{N}_P$ and $\mathbf{N}_E$ are countably infinite sets representing protocol and adversary nonces, respectively. (f/n means f has arity n.) Intuitively, encryption, decryption, verification, and signing keys are represented as $ek(r), dk(r), vk(r), sk(r)$ with a nonce r (the randomness used when generating the keys). $enc(ek(r'), m, r)$ encrypts m using the encryption key $ek(r')$ and randomness r. $sig(sk(r'), m, r)$ is a signature of m using the signing key $sk(r')$ and randomness r. The constructors $string_0$, $string_1$, and $empty$ are used to model arbitrary strings used as payload in a protocol (e.g., a bitstring 010 would be encoded as $string_0(string_1(string_0(empty)))$). $garbage$, $garbageEnc$, and $garbageSig$ are constructors necessary to express certain invalid terms the adversary may send, these constructors are not used by the protocol.

Message type.[2] We define $\mathbf{T}$ as the set of all terms T

[1] In an interesting recent work, Comon-Lundh *et al.* [20] also achieved a computational soundness result for dishonest keys. Their work is orthogonal to our work in that they proposed an extension of the symbolic model while keeping the standard security assumptions IND-CPA and IND-CTXT for the encryption scheme. As explained before, we avoid symbolic extensions at the cost of novel cryptographic definitions.

[2] In the CoSP framework, the message type represents the set of all well-formed terms. Having such a restriction (and excluding, e.g., $enc(dk(N), \dots)$ or similar) makes life easier. However, when applying the computational soundness result to a calculus that does not support message types, one needs to remove the restriction that only terms in the message type are considered. [4] give a theorem that guarantees that this can be done without losing computational soundness.

matching the following grammar:

$$\begin{aligned} T ::= {} & \mathit{enc}(\mathit{ek}(N), T, N) \mid \mathit{ek}(N) \mid \mathit{dk}(N) \mid \\ & \mathit{sig}(\mathit{sk}(N), T, N) \mid \mathit{vk}(N) \mid \mathit{sk}(N) \mid \\ & \mathit{pair}(T, T) \mid S \mid N \mid \\ & \mathit{garbage}(N) \mid \mathit{garbageEnc}(T, N) \mid \\ & \mathit{garbageSig}(T, N) \\ S ::= {} & \mathit{empty} \mid \mathit{string}_0(S) \mid \mathit{string}_1(S) \end{aligned}$$

where the nonterminal N stands for nonces.

Destructors. $\mathbf{D} := \{\mathit{dec}/2, \mathit{isenc}/1, \mathit{isek}/1, \mathit{isdk}/1, \mathit{ekof}/1, \mathit{ekofdk}/1, \mathit{verify}/2, \mathit{issig}/1, \mathit{isvk}/1, \mathit{issk}/1, \mathit{vkof}/2, \mathit{vkofsk}/1, \mathit{fst}/1, \mathit{snd}/1, \mathit{unstring}_0/1, \mathit{unstring}_1/1, \mathit{equals}/2\}$. The destructors *isek*, *isdk*, *isvk*, *issk*, *isenc*, and *issig* realize predicates to test whether a term is an encryption key, decryption key, verification key, signing key, ciphertext, or signature, respectively. *ekof* extracts the encryption key from a ciphertext, *vkof* extracts the verification key from a signature. $\mathit{dec}(\mathit{dk}(r), c)$ decrypts the ciphertext c. $\mathit{verify}(\mathit{vk}(r), s)$ verifies the signature s with respect to the verification key $\mathit{vk}(r)$ and returns the signed message if successful. *ekofdk* and *vkofsk* compute the encryption/verification key corresponding to a decryption/signing key. The destructors *fst* and *snd* are used to destruct pairs, and the destructors $\mathit{unstring}_0$ and $\mathit{unstring}_1$ allow to parse payload-strings. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, $\mathit{unstring}_i$, and $\mathit{equals}(\cdot, \mathit{empty})$.)

The destructors are defined by the rules in Figure 1; an application matching none of these rules evaluates to $\bot$:

Deduction relation. $\vdash$ is the smallest relation satisfying the rules in Figure 2. This deduction relation specifies which terms the adversary can deduce given already known messages S. We use the shorthand eval_f for the application of a constructor or destructor. $\mathrm{eval}_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ if $f(t_1, \ldots, t_n) \neq \bot$ and $f(t_1, \ldots, t_n) \in \mathbf{T}$ and $\mathrm{eval}_f(t_1, \ldots, t_n) = \bot$ otherwise.

Protocols. We use the protocol model from the CoSP framework [4]. There, a protocol is modeled as a (possibly infinite) tree of nodes. Each node corresponds to a particular protocol action such as receiving a term from the adversary, sending a previously computed term to the adversary, applying a constructor or destructor to previously computed terms (and branching depending on whether the application is successful), or picking a nonce. We do not describe the protocol model in detail here, but it suffices to know that a protocol can freely apply constructors and destructors (computation nodes), branch depending on destructor success, and communicate with the adversary. Despite the simplicity of the model, it is expressive enough to embed powerful calculi such as the applied π-calculus (shown in [4]) or RCF, a core calculus for F# (shown in [6]).

Protocol execution. Given a particular protocol Π (modeled as a tree), the set of possible protocol traces is defined by traversing the tree: in case of an input node the adversary nondeterministically picks a term t with $S \vdash t$ where S are the terms sent so far through output nodes; at computation nodes, a new term is computed by applying a constructor or destructor to terms computed/received at earlier nodes; then the left or right successor is taken depending on whether the destructor succeeded. The sequence of nodes we traverse in this fashion is called a *symbolic node trace* of the protocol. By specifying sets of node traces, we can specify trace properties for a given protocol. We refer to [4] for details on the protocol model and its semantics.

$$\begin{aligned}
\mathit{dec}(\mathit{dk}(t_1), \mathit{enc}(\mathit{ek}(t_1), m, t_2)) &= m \\
\mathit{isenc}(\mathit{enc}(\mathit{ek}(t_1), t_2, t_3)) &= \mathit{enc}(\mathit{ek}(t_1), t_2, t_3) \\
\mathit{isenc}(\mathit{garbageEnc}(t_1, t_2)) &= \mathit{garbageEnc}(t_1, t_2) \\
\mathit{isek}(\mathit{ek}(t)) &= \mathit{ek}(t) \\
\mathit{isdk}(\mathit{dk}(t)) &= \mathit{dk}(t) \\
\mathit{ekof}(\mathit{enc}(\mathit{ek}(t_1), m, t_2)) &= \mathit{ek}(t_1) \\
\mathit{ekof}(\mathit{garbageEnc}(t_1, t_2)) &= t_1 \\
\mathit{ekofdk}(\mathit{dk}(t_1)) &= \mathit{ek}(t_1) \\
\mathit{verify}(\mathit{vk}(t_1), \mathit{sig}(\mathit{sk}(t_1), t_2, t_3)) &= t_2 \\
\mathit{issig}(\mathit{sig}(\mathit{sk}(t_1), t_2, t_3)) &= \mathit{sig}(\mathit{sk}(t_1), t_2, t_3) \\
\mathit{issig}(\mathit{garbageSig}(t_1, t_2)) &= \mathit{garbageSig}(t_1, t_2) \\
\mathit{isvk}(\mathit{vk}(t_1)) &= \mathit{vk}(t_1) \\
\mathit{issk}(\mathit{sk}(t)) &= \mathit{sk}(t) \\
\mathit{vkof}(\mathit{sig}(\mathit{sk}(t_1), t_2, t_3)) &= \mathit{vk}(t_1) \\
\mathit{vkof}(\mathit{garbageSig}(t_1, t_2)) &= t_1 \\
\mathit{vkofsk}(\mathit{sk}(t_1)) &= \mathit{vk}(t_1) \\
\mathit{fst}(\mathit{pair}(x, y)) &= x \\
\mathit{snd}(\mathit{pair}(x, y)) &= y \\
\mathit{unstring}_0(\mathit{string}_0(s)) &= s \\
\mathit{unstring}_1(\mathit{string}_1(s)) &= s \\
\mathit{equals}(t_1, t_1) &= t_1
\end{aligned}$$

Figure 1: Rules defining the destructors. A destructor application matching none of these rules evaluates to $\bot$.

3. DEFINITIONS OF COMPUTATIONAL SOUNDNESS

We now sketch how computational soundness is defined. For details, we refer to [4]. In order to say whether we have computational soundness or not, we first need to specify a computational implementation A. Following [4], this is done by specifying a partial deterministic function $A_F : (\{0,1\}^*)^n \to \{0,1\}^*$ for each constructor or destructor F/n.[3] Also A_N is an distribution of bitstrings modeling the distribution of nonces. Given a computational implementation, we can execute a protocol in the computational model. This execution is fully analogous to the symbolic execution, except that in computation nodes, instead of applying constructors/destructors F to terms, we apply A_F to bitstrings, and in input/output nodes, we receive/send bitstring from/to a polynomial-time adversary.

Definition 1 (Comput. soundn. – simplified [4]). *We say a computational implementation A is a computationally sound implementation of a symbolic model for a*

[3]Probabilistic algorithms such as encryption are modeled by an explicit additional argument that takes a nonce as randomness.

$$\frac{m \in S}{S \vdash m} \qquad \frac{N \in \mathbf{N}_E}{S \vdash N} \qquad \frac{S \vdash \underline{t} \quad \underline{t} \in \mathbf{T} \quad F \in \mathbf{C} \cup \mathbf{D} \quad \mathrm{eval}_F(\underline{t}) \neq \bot}{S \vdash \mathrm{eval}_F(\underline{t})}$$

Figure 2: Deduction rules for the symbolic model

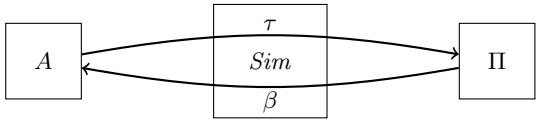

Figure 3: A typical CoSP simulator

class P of protocols if the following holds with overwhelming probability for any polynomial-time adversary A and any protocol $\Pi \in P$: The node trace in the computational protocol execution is a valid node trace in the symbolic protocol execution.

4. COMPUTATIONAL SOUNDNESS PROOFS IN COSP

Before we proceed and present the computational assumptions, we first give an overview on how prior computational soundness proofs were conducted. Since we based our result on the proof in the CoSP framework, we review the proof as it was performed there [4]. The problems we will face are not specific to their proof though.

Remember that in the CoSP framework, a protocol is modeled as a tree whose nodes correspond to the steps of the protocol execution; security properties are expressed as sets of node traces. Computational soundness means that for any polynomial-time adversary A the trace in the computational execution is, except with negligible probability, also a possible node trace in the symbolic execution. The approach for showing this is to construct a so-called simulator *Sim*. The simulator is a machine that interacts with a symbolic execution of the protocol Π on the one hand, and with the adversary A on the other hand; we call this a hybrid execution. (See Figure 3.) The simulator has to satisfy the following two properties:

- Indistinguishability: The node trace in the hybrid execution is computationally indistinguishable from that in the computational execution with adversary A.
- Dolev-Yaoness: The simulator *Sim* never (except for negligible probability) sends terms t to the protocol with $S \nvdash t$ where S is the list of terms *Sim* received from the protocol so far.

The existence of such a simulator (for any A) then guarantees computational soundness: Dolev-Yaoness guarantees that only node traces occur in the hybrid execution that are possible in the symbolic execution, and indistinguishability guarantees that only node traces occur in the computational execution that can occur in the hybrid one.

How to construct a simulator? In [4], the simulator *Sim* is constructed as follows: Whenever it gets a term from the protocol, it constructs a corresponding bitstring and sends it to the adversary, and when receiving a bitstring from the adversary it parses it and sends the resulting term to the protocol. Constructing bitstrings is done using a function β, parsing bitstrings to terms using a function τ. (See Figure 3.) The simulator picks all random values and keys himself: For each protocol nonce N, he initially picks a bitstring r_N. He then translates, e.g., $\beta(N) := r_N$ and $\beta(ek(N)) := A_{ek}(r_N)$ and $\beta(enc(ek(N), t, M)) := A_{enc}(A_{ek}(r_N), \beta(t), r_M)$. Translating back also is natural: Given $m = r_N$, we let $\tau(m) := N$, and if c is a ciphertext that can be decrypted as m using $A_{dk}(r_N)$, we set $\tau(c) := enc(ek(N), \tau(m), M)$. However, in the last case, a subtlety occurs: what nonce M should we use as symbolic randomness in $\tau(c)$? Here we distinguish two cases:

If c was earlier produced by the simulator: Then c was the result of computing $\beta(t)$ for some $t = enc(ek(N), t', M)$ and some nonce M. We then simply set $\tau(c) := t$ and have consistently mapped c back to the term it came from.

If c was not produced by the simulator: In this case it is an adversary generated encryption, and M should be an adversary nonce to represent that fact. We could just use a fresh nonce $M \in \mathbf{N}_E$, but that would introduce the need of additional bookkeeping: If we compute $t := \tau(c)$, and later $\beta(t)$ is invoked, we need to make sure that $\beta(t) = c$ in order for the *Sim* to work consistently (formally, this is needed in the proof of the indistinguishability of *Sim*). And we need to make sure that when computing $\tau(c)$ again, we use the same M. This bookkeeping can be avoided using the following trick: We identify the adversary nonces with symbols N^m annotated with bitstrings m. Then $\tau(c) := enc(ek(N), \tau(m), N^c)$, i.e., we set $M := N^c$. This ensures that different c get different randomness nonces N^c, the same c is always assigned the same N^c, and $\beta(t)$ is easy to define: $\beta(enc(ek(N), m, N^c)) := c$ because we know that $enc(ek(N), m, N^c)$ can only have been produced by $\tau(c)$. To illustrate, here are excerpts of the definitions of β and τ (the first matching rule counts):

- $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}, N \in \mathbf{N}_P$
- $\tau(c) := enc(ek(M), \tau(m), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(M)$ for some $M \in \mathbf{N}_P$ and $m := A_{dec}(A_{dk}(r_M), c) \neq \bot$
- $\beta(enc(ek(N), t, M)) := A_{enc}(A_{ek}(r_N), \beta(t), r_M)$ if $M \in \mathbf{N}_P$
- $\beta(enc(ek(M), t, N^m)) := m$ if $M \in \mathbf{N}_P$

Bitstrings m that cannot be suitably parsed are mapped into terms $garbage(N^m)$ and similar that can then be mapped back by β using the annotation m.

Showing indistinguishability. Showing indistinguishability essentially boils down to showing that the functions β and τ consistently translate terms back and forth. More precisely, we show that $\beta(\tau(m)) = m$ and $\tau(\beta(t)) = t$. Furthermore, we need to show that in any protocol step where a constructor or destructor F is applied to terms $t_1, \ldots, t_n$, we have that $\beta(F(t_1, \ldots, t_n)) = A_F(\beta(t_1), \ldots, \beta(t_n))$. This makes sure that the computational execution (where A_F is applied) stays in sync with the hybrid execution (where F is applied and the result is translated using β). The proofs of

these facts are lengthy (involving case distinctions over all constructors and destructors) but do not provide much additional insight; they are very important though because they are responsible for most of the implementation conditions that are needed for the computational soundness result.

Showing Dolev-Yaoness. The proof of Dolev-Yaoness is where most of the actual cryptographic assumptions come in. In this sketch, we will slightly deviate from the original proof in [4] for easier comparison with the proof in the present paper. The differences are, however, inessential. Starting from the simulator Sim, we introduce a sequence of simulators Sim_2, Sim_4, Sim_7. (We use a numbering with gaps here to be compatible with our full proof [7].)

In Sim_2, we change the function β as follows: When invoked as $\beta(enc(ek(N), t, M))$ with $M \in \mathbf{N}_P$, instead of computing $A_{enc}(A_{ek}(r_N), \beta(t), r_M)$, β invokes an encryption oracle $\mathcal{O}^N_{enc}$ to produce the ciphertext c. Similarly, $\beta(ek(N))$ returns the public key provided by the oracle $\mathcal{O}^N_{enc}$. The hybrid executions of Sim and Sim_2 are then indistinguishable. (Here we use that the protocol conditions guarantee that no randomness is used in two places.) Also, the function τ is changed to invoke $\mathcal{O}^N_{enc}$ whenever it needs to decrypt a ciphertext while parsing. Notice that if c was returned by $\beta(t)$ with $t := enc(\dots)$, then $\tau(c)$ just recalls the term t without having to decrypt. Hence $\mathcal{O}^N_{enc}$ is never asked to decrypt a ciphertext it produced.

In Sim_4, we replace the encryption oracle $\mathcal{O}^N_{enc}$ by a fake encryption oracle $\mathcal{O}^N_{fake}$ that encrypts zero-plaintexts instead of the true plaintexts. Since $\mathcal{O}^N_{enc}$ is never asked to decrypt a ciphertext it produced, IND-CCA2 security guarantees that the hybrid executions of Sim_2 and Sim_4 are indistinguishable. Since the plaintexts given to $\mathcal{O}^N_{fake}$ are never used, we can further change $\beta(enc(N, t, M))$ to never even compute the plaintext $\beta(t)$.

Finally, in Sim_7, we additionally change β to use a signing oracle in order to produce signatures. As in the case of Sim_2, the hybrid executions of Sim_4 and Sim_7 are indistinguishable.

Since the hybrid executions of Sim and Sim_7 are indistinguishable, in order to show Dolev-Yaoness of Sim, it is sufficient to show Dolev-Yaoness of Sim_7.

The first step to showing this is to show that whenever Sim_7 invokes $\beta(t)$, then $S \vdash t$ holds (where S are the terms received from the protocol). This follows from the fact that β is invoked on terms t_0 sent by the protocol (which are then by definition in S), and recursively descends only into subterms that can be deduced from t_0. In particular, in Sim_4 we made sure that $\beta(t)$ is not invoked by $\beta(enc(ek(N), t, M))$; t would not be deducible from $enc(ek(N), t, M)$.

Next we prove that whenever $S \nvdash t$, then t contains a visible subterm t_{bad} with $S \nvdash t_{bad}$ such that t_{bad} is a protocol nonce, or a ciphertext $enc(\dots, N)$ where N is a protocol nonces, or a signature, or a few other similar cases. (Visibility is a purely syntactic condition and essentially means that t_{bad} is not protected by an honestly generated encryption.)

Now we can conclude Dolev-Yaoness of Sim_7: If it does not hold, Sim_7 sends a term $t = \tau(m)$ where m was sent by the adversary A. Then t has a visible subterm t_{bad}. Visibility implies that the recursive computation of $\tau(m)$ had a subinvocation $\tau(m_{bad}) = t_{bad}$. For each possible case of t_{bad} we derive a contradiction. For example, if t_{bad} is a protocol nonce, then $\beta(t_{bad})$ was never invoked (since $S \nvdash t_{bad}$) and thus $m_{bad} = r_N$ was guessed by the simulator without ever accessing r_N which can happen only with negligible probability. Other cases are excluded, e.g., by the unforgeability of the signature scheme and by the unpredictability of encryptions. Thus, Sim_7 is Dolev-Yao, hence Sim is indistinguishable and Dolev-Yao. Computational soundness follows.

5. RESTRICTIONS IN THE PROOF AND HOW TO SOLVE THEM

The proof of computational soundness from [4] only works if protocols obey the following restrictions:

- The protocol never sends a decryption key (not even within a ciphertext).
- The protocol never decrypts using a decryption key it received from the net.
- The protocol avoids key cycles (i.e., encryptions of decryption keys using their corresponding encryptions keys). This latter condition is actually already ensured by never sending decryption keys, but we mention it explicitly for completeness.

(Similar restrictions occur for signing keys in [4], however, those restrictions are not due to principal issues, removing them just adds some cases to the proof.)

We will now explain where these restrictions come from and how we avoid them in our proof.

5.1 Sending secret keys

The first restriction that we encounter in the above proof is that we are not allowed to send secret keys. For example, the following simple protocol is not covered by the above proof:

Alice picks a encryption/decryption key pair (ek, dk) and publishes ek. Then Alice sends $enc(ek, N)$ for some fresh nonce N. And finally Alice sends dk.

When applying the above proof to this protocol, the faking simulator (more precisely, the function τ in that simulator) will translate $enc(ek, N)$ into an encryption c of 0 (as opposed to an encryption of r_N). But then, when dk is sent later by the symbolic protocol, the simulator would have to send the corresponding computational decryption key. But that would allow the adversary to decrypt c, and the adversary would notice that c is a fake ciphertext.

The following solution springs to mind: We modify the faking simulator such that he will only produce fake ciphertexts when encrypting with respect to a key pair whose secret key will never be revealed. Indeed, if we could do so, it might solve our problem. However, in slightly more complex protocols than our toy example, the simulator may not know in advance whether a given secret key will be revealed (this may depend on the adversary's actions which in turn may depend on the messages produced by the simulator). Of course, we might let the simulator guess which keys will be revealed. That, however, will only work when the number of keys is logarithmic in the security parameter. Otherwise the probability of guessing correctly will be negligible.[4]

(Notice also that the problem is also not solved if the simulator does not produce fake ciphertexts if in doubt: Then

[4]This is closely related to selective opening security (SOA) [14]. However, although selective SOA addresses a similar problem, it is not clear how SOA could be used to prove computational soundness.

our argument that the bitstring m_{bad} is unguessable would become invalid.)

To get rid of the restriction, we take a different approach. Instead of forcing the simulator to decide right away whether a given ciphertext should be a fake ciphertext or not, we let him decide this later. More precisely, we make sure that the simulator can produce a ciphertext c without knowing the plaintext, and later may "reprogram" the ciphertext c such that it becomes an encryption of a message m of his choice. (But not after revealing the secret key, of course.)

At the first glance, this seems impossible. Since the ciphertext c may already have been sent to the adversary, c cannot be changed. It might be possible to have an encryption scheme where for each encryption key, there can be many decryption keys; then the simulator could produce a special decryption key that decrypts c to whatever he wishes. But simple counting arguments show that then the decryption key would need to be as long as the plaintexts of all ciphertexts c produced so far together. This would lead to a highly impractical scheme, and be impossible if we do not impose an a-priori bound on the number of ciphertexts. (See [32].)

However, we can get around this impossibility if we work in the random oracle model. (In the following, we use the word random oracle for any oracle chosen uniformly out of a family of functions; thus also the ideal cipher model or the generic group model fall under this term. The "standard" random oracle [15] which is a uniformly randomly chosen function from the set of all functions we call "random hash oracle" for disambiguation.)

In the random oracle model, we can see the random oracle as a function that is initially undefined, and upon access, the function table is populated as needed (lazy sampling). This enables the following proof technique: When a certain random oracle location has not been queried yet, we may set it to a particular value of our choosing (this is called "programming the random oracle"). In our case this can be used to program a ciphertext c: As long as we make sure that the adversary has not yet queried the random oracle at the locations needed for decrypting c (e.g., because to find these locations he needs to know the secret key), we can still change the value of the oracle at these locations. This in turn may allow us to change the value that c decrypts to.

Summarizing, we look for an encryption scheme with the following property: There is a strategy for producing (fake) keys and ciphertexts, and for reprogramming the random oracle (we will call this strategy the "ciphertext simulator"), such that the following two things are indistinguishable: (a) (Normally) encrypting a value m, sending the resulting ciphertext c, and then sending the decryption key. (b) Producing a fake ciphertext c. Choosing m. And sending the decryption key.

Such a scheme could then be used in our computational soundness proof: Sim_2 would encrypt messages m normally. Sim_4 would produce fake ciphertexts c instead, and only when revealing the decryption key, reprogram the ciphertexts c to contain the right messages m. Then, we would consider an additional simulator Sim_5 that does not even compute m until it is needed. This will then allow us to argue that the bitstring m_{bad} corresponding to a "bad" subterm t_{bad} cannot be guessed because the information needed for guessing this bitstring was never computed/accessed.

A security definition for encryption schemes with the required properties has been presented in [35] (called PROG-KDM), together with a natural construction satisfying the definition. In the following, we present and explain their definition and how it allows us to get computational soundness for protocols sending secret keys.

Formally defining PROG-KDM security turns out to be more complex than one might expect. We cannot just state that the ciphertext simulator is indistinguishable from an honest encryption oracle. The ciphertext simulator has a completely different interface from the honest encryption oracle. In particular, it expects the plaintext when being asked for the secret key, while the encryption oracle would expect these upon encryption. To cope with this problem, we define two "wrappers", the real and the fake challenger. The real challenger essentially gives us access to the encryption algorithm while the fake challenger, although it expects the plaintexts during encryption (to be indistinguishable from the real challenger), uses the plaintexts only when the decryption key is to be produced. These two challengers should then be indistinguishable. (The challengers additionally make sure that the adversary does not perform any forbidden queries such as submitting a ciphertext for decryption that was produced by the challenger.)

We first define the real challenger. The real challenger needs to allows us to query the encryption and decryption keys, to perform encryptions and decryptions, and to give us access to the underlying random oracle. However, if we only have these queries, situations like the following would lead to problems: The adversary wishes to get $\mathsf{Enc}(ek_1, \mathsf{Enc}(ek_2, m))$. We do not wish the adversary to have to request $\mathsf{Enc}(ek_2, m)$ first and then resubmit it for the second encryption, because this would reveal $\mathsf{Enc}(ek_2, m)$, and we might later wish to argue that $\mathsf{Enc}(ek_2, m)$ stays secret. To be able to model such setting, we need to allow the adversary to evaluate sequences of queries without revealing their outcome. For this, we introduce queries such as $R := \mathtt{enc}_{\mathtt{ch}}(N, R_1)$. This means: Take the value from register R_1, encrypt it with the key with index $N \in \{0,1\}^*$, and store the result in register R. Also, we need a query to apply arbitrary functions to registers: $R := \mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$ applies the circuit C to registers $R_1, \ldots, R_n$. (This in particular allows us to load a fixed value into a register by using a circuit with zero inputs ($n = 0$). Finally, we have a query $\mathtt{reveal}_{\mathtt{ch}}(R_1)$ that outputs the content of a register.

Formally, the definition of the real challenger is the following:

Definition 2 (Real challenger). *Fix an oracle $\mathcal{O}$ and an encryption scheme (K, E, D) relative to that oracle. The* real challenger RC *is an interactive machine defined as follows.* RC *has access to the oracle $\mathcal{O}$.* RC *maintains a family $(ek_N, dk_N)_{N\in\{0,1\}^*}$ of key pairs (initialized as $(ek_N, dk_N) \leftarrow K(1^\eta)$ upon first use), a family $(reg_N)_{N\in\{0,1\}^*}$ of registers (initially all $reg_N = \bot$), and a family of sets $cipher_N$ (initially empty).* RC *responds to the following queries (when no answer is specified, the empty word is returned):*

- $R := \mathtt{getek}_{\mathtt{ch}}(N)$*:* RC *sets $reg_R := ek_N$.*
- $R := \mathtt{getdk}_{\mathtt{ch}}(N)$*:* RC *sets $reg_R := dk_N$.*
- $R := \mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$ *where C is a Boolean circuit:*[5] *Compute $m := C(reg_{R_1}, \ldots, reg_{R_n})$ and set $reg_R := m$.*

[5]Note that from the description of a circuit, it is possible to

- $R := \mathtt{enc}_{\mathtt{ch}}(N, R_1)$: *Compute* $c \leftarrow E^{\mathcal{O}}(ek_N, reg_{R_1})$, *append* c *to* $cipher_N$, *and set* $reg_R := c$.
- $\mathtt{oracle}_{\mathtt{ch}}(x)$: *Return* $\mathcal{O}(x)$.
- $\mathtt{dec}_{\mathtt{ch}}(N, c)$: *If* $c \in cipher_N$, *return* `forbidden` *where* `forbidden` *is a special symbol (different from any bitstring and from a failed decryption* $\bot$*). Otherwise, invoke* $m \leftarrow D^{\mathcal{O}}(dk_N, c)$ *and return* m.
- $\mathtt{reveal}_{\mathtt{ch}}(R_1)$: *Return* reg_{R_1}.

Here N *and* c *range over bitstrings,* R *ranges over bitstrings with* $reg_R = \bot$ *and the* R_i *range over bitstrings* R *with* $reg_{R_i} \neq \bot$.

Notice that the fact that we can do "hidden evaluations" of complex expressions, also covers KDM security (security under key-dependent messages): We can make a register contain the computation of, e.g., $\mathsf{Enc}(ek, dk)$ where dk is the decryption key corresponding to ek.

We now proceed to define the fake challenger. The fake challenger responds to the same queries, but computes the plaintexts as late as possible. In order to do this, upon a query such as $R := \mathtt{enc}_{\mathtt{ch}}(N, R_1)$, the fake challenger just stores the symbolic expression "$\mathtt{enc}_{\mathtt{ch}}(N, R_1)$" in register R (instead of an actual ciphertext). Only when the content of a register is to be revealed, the bitstrings are recursively computed (using the function `FCRetrieve` below) by querying the ciphertext simulator. Thus, before defining the fake challenger, we first have to define formally what a ciphertext simulator is:

Definition 3 (Ciphertext simulator). *A* ciphertext simulator CS *for an oracle* $\mathcal{O}$ *is an interactive machine that responds to the following queries:* $\mathtt{fakeenc}_{\mathtt{cs}}(R, l)$, $\mathtt{dec}_{\mathtt{cs}}(c)$, $\mathtt{enc}_{\mathtt{cs}}(R, m)$, $\mathtt{getek}_{\mathtt{cs}}()$, $\mathtt{getdk}_{\mathtt{cs}}()$, *and* $\mathtt{program}_{\mathtt{cs}}(R, m)$. *Any query is answered with a bitstring (except* $\mathtt{dec}_{\mathtt{cs}}(c)$ *which may also return* $\bot$*). A ciphertext simulator runs in polynomial-time in the total length of the queries. A ciphertext simulator is furthermore given access to an oracle* $\mathcal{O}$. *The ciphertext simulator is also allowed to program* $\mathcal{O}$ *(that is, it may perform assignments of the form* $\mathcal{O}(x) := y$*). Furthermore, the ciphertext simulator has access to the list of all queries made to* $\mathcal{O}$ *so far.*[6]

The interesting queries here are $\mathtt{fakeenc}_{\mathtt{cs}}(R, l)$ and $\mathtt{program}_{\mathtt{cs}}(R, m)$. A $\mathtt{fakeenc}_{\mathtt{cs}}(R, l)$-query is expected to return a fake ciphertext for an unspecified plaintext of length l (associated with a handle R). And a subsequent $\mathtt{program}_{\mathtt{cs}}(R, m)$-query with $|m| = l$ is supposed to program the random oracle such that decrypting c will return m. The ciphertext simulator expects to get all necessary $\mathtt{program}_{\mathtt{cs}}(R, m)$-queries directly after a $\mathtt{getdk}_{\mathtt{cs}}()$-query revealing the key. (Formally, we do not impose this rule, but the PROG-KDM does not guarantee anything if the ciphertext simulator is not queried in the same way as does the fake challenger below.) We stress that we allow to first ask for the key and then to program. This is needed to handle key dependencies, e.g., if we wish to program the plaintext to be the decryption key. The definition of the fake challenger will make sure that although we reveal the decryption key before programming, we do not use its value for anything but the programming until the programming is done.

Note that we do not fix any concrete behavior of the ciphertext simulator since our definition will just require the existence of some ciphertext simulator.

We can now define the real challenger together with its recursive retrieval function `FCRetrieve`:

Definition 4 (Fake challenger). *Fix an oracle* $\mathcal{O}$, *a length-regular encryption scheme* (K, E, D) *relative to that oracle, and a ciphertext simulator* CS *for* $\mathcal{O}$. *The* fake challenger FC *for* CS *is an interactive machine defined as follows.* FC *maintains the following state:*

- *A family of instances* $(\mathrm{CS}_N)_{N\in\{0,1\}^*}$ *of* CS *(initialized upon first use). Each ciphertext simulator is given (read-write) oracle access to* $\mathcal{O}$.
- *A family* $(reg_R)_{R\in\{0,1\}^*}$ *of registers (initially all* $reg_R = \bot$*). Registers* reg_N *are either undefined (*$reg_N = \bot$*), or bitstrings, or queries (written "*$\mathtt{getek}_{\mathtt{ch}}(N)$*" or "*$\mathtt{getdk}_{\mathtt{ch}}(N)$*" or "*$\mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$*" etc.).*
- *A family* $(cipher_N)_{N\in\{0,1\}^*}$ *of sets of bitstrings. (Initially all empty.)*

FC *answers to the same queries as the real challenger, but implements them differently:*

- $R := \mathtt{getek}_{\mathtt{ch}}(N)$ *or* $R := \mathtt{getdk}_{\mathtt{ch}}(N)$ *or* $R := \mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$ *or* $R := \mathtt{enc}_{\mathtt{ch}}(N, R_1)$: *Set* $reg_R :=$ "$\mathtt{getek}_{\mathtt{ch}}(N)$" *or* $reg_R :=$ "$\mathtt{getdk}_{\mathtt{ch}}(N)$" *or* $reg_R :=$ "$\mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$" *or* $reg_R :=$ "$\mathtt{enc}_{\mathtt{ch}}(N, R_1)$", *respectively.*
- $\mathtt{dec}_{\mathtt{ch}}(N, c)$: *If* $c \in cipher_N$, *return* `forbidden`. *Otherwise, query* $\mathtt{dec}_{\mathtt{cs}}(c)$ *from* CS_N *and return its response.*
- $\mathtt{oracle}_{\mathtt{ch}}(x)$: *Return* $\mathcal{O}(x)$.
- $\mathtt{reveal}_{\mathtt{ch}}(R_1)$: *Compute* $m \leftarrow \mathtt{FCRetrieve}(R_1)$. (`FCRetrieve` *is defined below in Definition 5.) Return* m.

Definition 5 (Retrieve function of FC). *The* retrieve function `FCRetrieve` *has access to the registers* reg_R *and the ciphertext simulators* CS_N *of* FC. *It additionally stores a family* $(plain_N)_{N\in\{0,1\}^*}$ *of lists between invocations (all* $plain_N$ *are initially empty lists).* `FCRetrieve` *takes an argument* R *(with* $reg_R \neq \bot$*) and is recursively defined as follows:*

- *If* reg_R *is a bitstring, return* reg_R.
- *If* $reg_R =$ "$\mathtt{getek}_{\mathtt{ch}}(N)$": *Query* CS_N *with* $\mathtt{getek}_{\mathtt{cs}}()$. *Store the answer in* reg_R. *Return* reg_R.
- *If* $reg_R =$ "$\mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$": *Compute* $m_i := \mathtt{FCRetrieve}(R_i)$ *for* $i = 1, \ldots, n$. *Compute* $m' := C(m_1, \ldots, m_n)$. *Set* $reg_R := m'$. *Return* m'.
- *If* $reg_R =$ "$\mathtt{enc}_{\mathtt{ch}}(N, R_1)$" *and there was no* $\mathtt{getdk}_{\mathtt{cs}}()$*-query to* CS_N *yet: Compute* $l := \mathtt{FCLen}(R_1)$. (`FCLen` *is defined in Definition 7 below.) Query* CS_N *with* $\mathtt{fakeenc}_{\mathtt{cs}}(R, l)$. *Denote the answer with* c. *Set* $reg_R := c$. *Append* $(R \mapsto R_1)$ *to the list* $plain_N$. *Append* c *to* $cipher_N$. *Return* c.
- *If* $reg_R =$ "$\mathtt{enc}_{\mathtt{ch}}(N, R_1)$" *and there was a* $\mathtt{getdk}_{\mathtt{cs}}()$*-query to* CS_N: *Compute* $m := \mathtt{FCRetrieve}(R_1)$. *Query* CS_N *with* $\mathtt{enc}_{\mathtt{cs}}(R, m)$. *Denote the answer with* c. *Set* $reg_R := c$. *Append* $(R \mapsto R_1)$ *to* $plain_N$. *Append* c *to* $cipher_N$. *Return* c.
- *If* $reg_R =$ "$\mathtt{getdk}_{\mathtt{ch}}(N)$": *Query* CS_N *with* $\mathtt{getdk}_{\mathtt{cs}}()$. *Store the answer in* reg_R. *If this was the first* $\mathtt{getdk}_{\mathtt{cs}}(N)$*-query for that value of* N, *do the following*

determine the length of its output. This will be important in the definition of `FCLen` below.

[6] Our scheme will not make use of the list of the queries to $\mathcal{O}$, but for other schemes this additional power might be helpful.

for each $(R' \mapsto R'_1) \in plain_N$ *(in the order they occur in the list):*
– *Invoke* $m := \texttt{FCRetrieve}(R'_1)$.
– *Send the query* $\texttt{program}_{\texttt{cs}}(R', m)$ *to* CS_N.
Finally, return reg_R.

The retrieve function uses the auxiliary function FCLen that computes what length a bitstring associated with a register should have. This function only makes sense if we require the encryption scheme to be length regular, i.e., the length of the output of the encryption scheme depends only on the lengths of its inputs.

Definition 6 (Length regular encryption scheme). *An encryption scheme* (K, E, D) *is* length-regular *if there are functions* $\ell_{ek}, \ell_{dk}, \ell_c$ *such that for all* $\eta \in \mathbb{N}$ *and all* $m \in \{0,1\}^*$ *and for* $(ek, dk) \leftarrow K(1^\eta)$ *and* $c \leftarrow E(ek, m)$ *we have* $|ek| = \ell_{ek}(\eta)$ *and* $|dk| = \ell_{dk}(\eta)$ *and* $|c| = \ell_c(\eta, |m|)$ *with probability* 1.

Definition 7 (Length function of FC). *The* length function FCLen *has (read-only) access to the registers* reg_R *of* FC. FCLen *takes an argument* R *(with* $reg_R \neq \perp$*) and is recursively defined as follows:*

- *If* reg_R *is a bitstring, return* $|reg_R|$.
- *If* $reg_R =$ *"*$\texttt{eval}_{\texttt{ch}}(C, R_1, \ldots, R_n)$*": Return the length of the output of the circuit* C. *(Note that the length of the output of a Boolean circuit is independent of its arguments.)*
- *If* $reg_R =$ *"*$\texttt{getek}_{\texttt{ch}}(N)$*" or* $reg_R =$ *"*$\texttt{getdk}_{\texttt{cs}}(N)$*": Let* ℓ_{ek} *and* ℓ_{dk} *be as in Definition 6. Return* $\ell_{ek}(\eta)$ *or* $\ell_{dk}(\eta)$*, respectively.*
- *If* $reg_R =$ *"*$\texttt{enc}_{\texttt{ch}}(N, R_1)$*": Let* ℓ_c *be as in Definition 6. Return* $\ell_c(\eta, \texttt{FCLen}(R_1))$.

We are now finally ready to define PROG-KDM security:

Definition 8 (PROG-KDM security). *A length-regular encryption scheme* (K, E, D) *(relative to an oracle* $\mathcal{O}$*) is PROG-KDM secure iff there exists a ciphertext simulator* CS *such that for all polynomial-time oracle machines* $\mathcal{A}$,[7] $\Pr[\mathcal{A}^{\mathrm{RC}}(1^\eta) = 1] - \Pr[\mathcal{A}^{\mathrm{FC}}(1^\eta) = 1]$ *is negligible in* η. *Here* RC *is the real challenger for* (K, E, D) *and* $\mathcal{O}$ *and* FC *is the fake challenger for* CS *and* $\mathcal{O}$. *Notice that* $\mathcal{A}$ *does not directly query* $\mathcal{O}$.

If we assume that the computational implementation of ek, dk, enc, dec is a PROG-KDM secure encryption scheme, we can make the proof sketched in Section 4 go through even if the protocol may reveal its decryption keys: The simulator Sim_2 uses the real challenger to produce the output of β. He does this by computing all of $\beta(t)$ inside the real challenger (using queries such as $R := \texttt{eval}_{\texttt{ch}}(C, \ldots)$). Then Sim_4 uses the fake challenger instead. By PROG-KDM security, Sim_2 and Sim_4 are indistinguishable. But Sim_4 still provides all values needed in the computation early (because the real challenger needs them early). But we can then define Sim_5 which does not use the real challenger any more, but directly accesses the ciphertext simulator (in the same way as the fake challenger would). Sim_5 is then indistinguishable from Sim_2, but, since the fake challenger performed all computations on when needed, Sim_2 now also performs all computations only when actually needed. This has the effect that in the end, we can show that the bitstring m_{bad} represents a contradiction because it guesses values that were never accessed.

[35] shows that PROG-KDM security can be achieved using a standard construction, namely hybrid encryption using any CCA2-secure key encapsulation mechanism, a block cipher (modeled as an ideal cipher) in CBC-mode, and encrypt-then-MAC with an arbitrary one-time MAC.

We have now removed the restriction that a protocol may not send its decryption keys. (And in one go, we also enabled key-cycles because PROG-KDM covers that case, too.) It remains to remove the restriction that we cannot use decryption keys received from the adversary,

The need for PROG-KDM security. The question that arises in this context is whether we actually need such a strong notion as PROG-KDM in this context. Obviously, IND-CCA2 security alone is not sufficient, there are schemes that are IND-CCA2 secure and break down in the presence of key-cycles.[8] But what about, e.g., KDM-CCA2 [18] that covers key dependent messages and active attacks?

To illustrate the necessity of a notion stronger than KDM-CCA2, consider the following example: Assume a protocol in which we want to share a secret s with n parties in such a way that $n/2$ parties are needed to recover the secret s. We do this by distributing n decryption keys to the n parties, and by producing a number of nested encryptions such that $n/2 - 1$ of the decryption keys are not sufficient to recover s. More precisely, we use the following protocol:[9]

- The dealer D chooses a nonce s and n key pairs (ek_i, dk_i).
- D chooses additional key pairs $(ek_{i,j}, dk_{i,j})$ for $i = 0, \ldots, n/2$ and $j = 0, \ldots, n$.
- D computes $e_{i,j} \leftarrow \mathsf{Enc}(ek_j, (\mathsf{Enc}(ek_{i-1,0}, dk_{i,j}), \ldots, \mathsf{Enc}(ek_{i-1,j-1}, dk_{i,j})))$ for all $i = 1, \ldots, n/2$, $j = 1, \ldots, n$, and publishes all $e_{i,j}, dk_{0,j}$. ($dk_{i,j}$ can then be computed if dk_j is known and at least i keys from $dk_1, \ldots, dk_j$ are known.)
- D computes $e_j \leftarrow \mathsf{Enc}(ek_{n/2,j}, s)$ for $j = 1, \ldots, n$, and publishes all e_j.
  (s can then be computed if $dk_{n/2,j}$ is known for some j. Thus, s can be computed if $n/2$ of the dk_j are known.)
- The adversary may choose $n/2 - 1$ indices $j \in \{1, \ldots, n\}$, and D sends dk_j for each of the selected j.
- The adversary wins if he guesses the secret nonce s.

It is easy to see that given $n/2$ keys dk_j, one can recover s. But in a reasonable symbolic model (e.g., the one from

[7] Here we consider $\mathcal{A}$ polynomial-time if it runs a polynomial number of steps in η, and the number of steps performed by RC or FC is also polynomially-bounded. This additional requirement is necessary since for an encryption scheme with multiplicative overhead (say, length-doubling), a sequence of queries $R_i := \texttt{enc}_{\texttt{ch}}(N, R_{i-1})$ of polynomial length will lead to the computation of an exponential-length ciphertext.

[8] Take, e.g., an IND-CCA2 secure encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and modify it such that $\mathsf{Enc}(ek, dk) := dk$ if ek and dk are a valid key pair, and let $\mathsf{Dec}(dk, dk) := dk$. It is easy to see that the modified scheme is still IND-CCA2 secure, but the key cycle $\mathsf{Enc}(ek, dk)$ reveals the decryption key.

[9] A simpler protocol would be to publish $e_I := \mathsf{Enc}(dk_{i_1}, \ldots \mathsf{Enc}(dk_{i_{n/2}}, s) \ldots)$ for each set $I = \{i_1, \ldots, i_{n/2}$ of size $n/2$. But that protocol would need to send an exponential number of ciphertexts I.

Section 2), the adversary cannot win.[10] So a computational soundness result without restrictions on sending and encrypting decryption keys would imply that the protocol is secure in the computational setting. Hence any security notion that allows us to derive the computational soundness result must also be sufficient to show that the protocol is secure in a computational setting. (Notice that situations similar to the one in this protocol could occur, e.g., if we enforce some complex authorization policy by a suitable set of nested encryptions.)

But it seems that IND-CCA2 or KDM-CCA2 security does not allow us to prove the security of this protocol. In a proof using one of these notions, one typically first defines a game G_1 which models an execution of the protocol. Then one defines a modified game G_2 in which some of the ciphertexts are replaced by encryptions of 0. Then one uses IND-CCA2 or KDM-CCA2 to show that G_1 and G_2 are indistinguishable. Finally, one uses that in game G_2, the secret s is never accessed, because we have replaced all occurrences of s by 0. If we would know in advance which keys dk_j the adversary requests, this proof would indeed go through. However, the selection of the dk_j by the adversary can be done adaptively, even depending on the values of the $e_{i,j}$. (E.g., the adversary could produce a hash of all protocol messages and use the bits in the hash value to decide which keys to reveal.) Hence, when encrypting, we do not know yet which ciphertexts will be opened. Since there are an exponential number of possibilities, we cannot guess. There seems to be no other way of choosing which ciphertexts should be 0-encryptions. Because of this, IND-CCA2 and KDM-CCA2 seem unapplicable for this protocol.[11]

Also notions such as IND-SO-CPA and SIM-SO-CPA which are designed for situations with selective opening of ciphertexts (cf. [17]) do not seem to match this protocol. Possibly extensions of these notions might cover this case, but it is not clear what these extensions should look like (in particular if we extend the protocol such that some of the $e_{i,j}$ may depend on other $e_{i,j}$, e.g., by including the latter in some of the plaintexts of the former).

So, it seems that the only known security notion for encryption schemes that can show the security of the above protocol is PROG-KDM. Thus it is not surprising that we need to use PROG-KDM security in our proof.

5.2 Receiving decryption keys

The second restriction we face in the proof sketched in Section 4 is that a protocol is not allowed to receive decryption keys. This is due to the way the simulator *Sim* parses a bitstring into a term (using the function τ): When receiving a ciphertext c for which the decryption key d is known, *Sim* computes $\tau(c) := enc(ek(N^e), \tau(m), N^c)$ where m is the plaintext of c and e the corresponding encryption key. If d is not known (because c was produced by the adversary with respect to a key that the protocol did not pick), *Sim* computes $\tau(c) := garbageEnc(ek(N^e), N^c)$. Notice that in the latter case we are cheating: even though c may be a valid ciphertext (just with respect to an encryption key whose decryption key we do not know), we declare it to be an invalid ciphertext. But the fact that we will never use the decryption key saves us: we will never be caught in a lie. The situation is different if we receive decryption keys from the adversary. Then the adversary might first send c which we parse to $garbageEnc(ek(N^e), N^c)$. Then later he sends us the corresponding decryption key d which we parse to $dk(N^e)$. But then in the computational execution, decrypting c using d works, while in the hybrid execution, decrypting $garbageEnc(ek(N^e), N^c)$ necessarily fails.

So if we allow the protocol to receive decryption keys, we need to change the simulator so that it parses $\tau(c) := enc(ek(N^e), t, N^c)$ when receiving a valid ciphertext c, even if the he cannot decrypt c. But then, how should the simulator compute the term t? And for that matter, how should the simulator know that c is valid? (It might be invalid, and then should be parsed as $garbageEnc(ek(N^e), N^c)$.)

A solution for this problem has been proposed in the first revision of [5] (not contained in later versions!) but has not been applied there. The idea is to allow the simulator to partially parse terms (lazy simulator). That is, we allow the simulator to output terms that contain variables, and to only after the hybrid execution we ask the simulator to decide what terms these variables stand for.

In our case, we change the simulator such that when parsing a ciphertext c (corresponding to a key not picked by the simulator), the simulator just outputs $\tau(c) := x^c$. (Here we assume an infinite set of variables x indexed by ciphertexts.) And in the end, when the hybrid execution finished, the simulator outputs a "final substitution" φ that maps x^c to either $enc(N^e, \tau(m), N^c)$ if by the end of the execution the simulator has learned the corresponding decryption key and can compute the plaintext m, or to $garbageEnc(N^e, N^c)$ if the decryption key was not received or decryption fails.

Unfortunately, to make this go through, the simulator gets an additional tasks. In the original hybrid execution, terms sent to the protocol do not contain variables, and whenever we reach a computation node in the protocol, we can apply the constructor or destructor to the arguments of that node and compute the resulting new term. This is not possible any more. For example, what would be the output a *dec*-node with plaintext argument x^c? Thus, the hybrid execution will in this case just maintain a "destructor term", in which the destructors are not evaluated. (E.g., a node might then store the term $dec(dk(N^e), x^c)$.) That leaves the following problem: A computation node branches to its yes- or no-successor depending on whether constructor/destructor application succeeds or fails. But in the hybrid execution, the constructor/destructor application is not evaluated, we do not know whether it succeeds or fails. This leads to an additional requirement for the simulator: After each computation node in the hybrid execution, the simulator is asked a "question". This question consists of the destructor term that is computed at the current node, and the simulator has to answer yes or no, indicating whether the application should be considered to have succeeded or failed. (And then the yes- or no-successor of the current node is taken accordingly.)

In our case, to answer these questions, the simulator will just reduce the term as much as possible (by evaluating de-

[10] Proof sketch: Fix a set $I \subseteq \{dk_1, \ldots, dk_n\}$. Let $S := \{e_j, e_{i,j}, dk_{0,j}\} \cup I$. By induction over i, we have that $S \vdash dk_{i,j}$ implies $|I \cap \{dk_1, \ldots, dk_j\}| \geq i$. If $S \vdash s$ there is a j with $S \vdash dk_{n/2,j}$, and hence $|I| \geq |I \cap \{dk_1, \ldots, dk_j\}| \geq n/2$. So $S \vdash s$ only if $|I| \geq n/2$, i.e., the adversary can only recover s by requesting at least $n/2$ keys.

[11] Of course, this is no proof that these notions are indeed insufficient. But it shows that at least natural proof approaches fail. We expect that an impossibility result relative to some oracle can be proven but we have not done so.

structors), replace variables x^c by *enc*- or *garbageEnc*-terms wherever we already know the necessary keys, and make the "right" choices when destructors are applied to x^c. If all destructors succeed, the simulator answers yes. A large part of the full proof is dedicated to showing that this can be done in a consistent fashion.

In [5], it is shown that if a lazy simulator with the following four properties (sketched below) exists, then we have computational soundness:

- Indistinguishability: The hybrid and the computational execution are indistinguishable (in terms of the nodes passed through in execution).
- DY-ness: Let φ be the final substitution (output by the simulator at the end of the execution). Then in any step of the execution it holds that $S\varphi \vdash t\varphi$ where t is the term sent by the simulator to the protocol, and S is the set of the terms received by the protocol (note that although S, t may be destructor terms, $S\varphi$ and $t\varphi$ do not contain variables any more and thus reduce to regular terms without destructors).
- Consistency: For any question Q that was asked from the simulator, we have that the simulator answered yes iff evaluating $Q\varphi$ (which contains destructors but no variables) does not return $\bot$.
- Abort-freeness: The simulator does not abort.

In the proof we construct such a simulator and show all the properties above. (Indistinguishability is relatively similar to the case without lazy parsing, but needs some additional care because the invariants need to be formulated with respect to unevaluated destructor terms. DY-ness follows the same lines but becomes considerably more complicated.)

In the proof of DY-ness, it does, however, turn out that lazy sampling does not fully solve the problem of receiving decryption keys. In fact, PROG-KDM security alone is not sufficient to guarantee computational soundness in this case (and neither is IND-CCA2). We illustrate the problem by an example protocol:

Alice picks a key $ek(N)$, a nonce M and sends a ciphertext $c := enc(ek(N), M, R)$ over the network (i.e., to the adversary). Then Alice expects a ciphertext c^*. Then Alice sends $dk(N)$. Then Alice expects a secret key sk^*. Finally, Alice tests whether $dec(sk^*, c^*) = (M, M)$.

It is easy to see that in the symbolic model, this test will always fail. But in the computational setting, it is possible to construct encryption schemes with respect to which the adversary can produce c^*, sk^* such that this test succeeds: Start with a secure encryption scheme $(\mathsf{KeyGen}', \mathsf{Enc}', \mathsf{Dec}')$. Then let $\mathsf{KeyGen} := \mathsf{KeyGen}'$, and $\mathsf{Enc} := \mathsf{Enc}'$, but modify Dec' as follows: Given a secret key of the form $sk = (\mathtt{special}, m)$, and a ciphertext $c = (\mathtt{special})$, $\mathsf{Dec}(sk, c)$ outputs m. On other inputs, Dec behaves like Dec'. Now the adversary can break the above protocol by sending $sk^* := (\mathtt{special}, (M, M))$. Notice that if $(\mathsf{KeyGen}', \mathsf{Enc}', \mathsf{Dec}')$ was PROG-KDM (or IND-CCA2), then $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is still PROG-KDM (or IND-CCA2): Both definitions say nothing about the behavior of the encryption scheme for dishonestly generated keys.

Of course, the above encryption scheme can easily be excluded by adding simple conditions on encryption schemes: Encryption keys should uniquely determine decryption keys and vice versa, any valid decryption key should successfully decrypt any ciphertext that was honestly generated using the corresponding encryption key, ciphertexts should determine their encryption key.

But even then a more complex construction works: Let $\mathcal{C}$ be some class of circuits such that for each $C \in \mathcal{C}$, there exists at most one x, y such that $C(x, y) = 1$. Let $\mathsf{KeyGen} := \mathsf{KeyGen}'$. Modify Enc' as follows: Upon input $ek = (\mathtt{special}, ek', C)$, $\mathsf{Enc}(ek, m)$ runs $\mathsf{Enc}'(ek', m)$. For other inputs, Enc behaves like Enc'. And Dec' is modified as follows: Upon input $dk = (\mathtt{special}, dk', C, x, y)$ and $c = (\mathtt{special}, ek', C)$ with $C(x, y) = 1$, $\mathsf{Dec}(dk, c)$ returns x. Upon $dk = (\mathtt{special}, dk', C, x, y)$ with $C(x, y) = 1$ and different c, $\mathsf{Dec}(dk, c)$ returns $\mathsf{Dec}'(dk', c)$. And upon all other inputs, Dec' behaves like Dec. Again, this construction does not loose PROG-KDM or IND-CCA2 security.

The adversary can break our toy protocol by choosing $\mathcal{C}$ as the class of circuits C_c defined by $C_c((M, M), sk) = 1$ if $\mathsf{Dec}(sk, c) = M$ and $C_c(x, y) = 0$ in all other cases. Then after getting c, the adversary chooses $(ek', dk') \leftarrow \mathsf{KeyGen}'$, $c^* := (special, ek', C_c)$ and after receiving a decryption key dk from Alice, he chooses $dk^* := (special, dk', C_c, (M, M), dk)$.

Notice that this example can be generalized to many different protocols where some m is uniquely determined by the messages sent by Alice, and the adversary learns m only after producing c but before sending the corresponding decryption key: Simply choose a different class $\mathcal{C}$ of circuits such that $C(m, x) = 1$ is a proof that m is the message encoded by Alice.

Clearly, the above example shows that PROG-KDM alone does not imply computational soundness. To understand what condition we need, let us first understand where the mismatch between the symbolic and the computational model is. In the symbolic model, the adversary can only produce an encryption of some message if he knows the underlying plaintext. In the computational model, however, even if we require unique decryption keys, it is sufficient that the underlying plaintext is fixed, it is not necessary that the adversary actually knows it.

Thus, to get computational soundness, we need to ensure that the adversary actually knows the plaintext of any message he produces. A common way for modeling knowledge is to require that we can extract the plaintext from the adversary. Since we work in the random oracle model anyway (as PROG-KDM only makes sense there), we use the following random-oracle based definition:[12]

DEFINITION 9. *We call an encryption scheme* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ malicious-key extractable *if for any polynomial-time* (A_1, A_2)*, there exists a polynomial-time algorithm MKE (the* malicious-key-extractor*such that the following probability is negligible:*

$$\Pr\big[\mathsf{Dec}^{\mathcal{O}}(d, c) \neq \bot \wedge \mathsf{Dec}^{\mathcal{O}}(d, c) \notin M : (z, c) \leftarrow A_1^{\mathcal{O}}(1^\eta), \\ M \leftarrow MKE^{\mathcal{O}}(1^\eta, c, queries), d \leftarrow A_2^{\mathcal{O}}(1^\eta, z)\big]$$

Here $\mathcal{O}$ is a random oracle. And queries is the list of all random oracle queries performed by A_1. And M is a list of messages (of polynomial length).

This definition guarantees that when the adversary pro-

[12]This is closely related to the notion of plaintext-awareness [16], except that plaintext-awareness applies only to the case of honestly generated keys.

duces a decryption key d that decrypts c to some message m, then he must already have known m while producing c.

Notice that malicious-key extractability is easy to achieve: Given a PROG-KDM secure encryption scheme, we modify it so that instead of encrypting m, we always encrypt $(m, H(m))$ where H is a random hash oracle (and decryption checks the correctness of that hash value). The resulting scheme does not loose PROG-KDM security and is malicious-key extractable.

In Definition 9, we only require that the extractor can output a list of plaintexts, one of which should be the correct one. We could strengthen the requirement and require the extractor to output only a single plaintext. This definition would considerably simplify our proof (essentially, we could get rid of lazy sampling since we can decrypt all adversary generated ciphertexts). However, that stronger definition would, for example, not be satisfied by the scheme that simply encrypts $(m, H(m))$. Since we strive for minimal assumptions, we opt for the weaker definition and the more complex proof instead.

How is malicious-key extractability used in the proof of computational soundness? We extend the simulator to call the extractor on all ciphertexts he sees (Sim_3). In the original proof, a simulator that is not DY implied that a term t with $S\varphi \not\vdash t\varphi$ is produced by τ in some step i. This means that $t\varphi$ has a "bad" subterm t_{bad}. This, however, does not immediately lead to a contradiction, because t_{bad} could be a subterm not of t, but of $\varphi(x^c)$ for some variable x^c in t. Since $\varphi(x^c)$ is produced at some later point, we cannot arrive at a contradiction (because the bitstring m_{bad} which is supposed to be unguessable in step i, might already have been sent in step j). But if the simulator runs the malicious-key extractor in step i, we can conclude that the bitstring m_{bad} corresponding to the subterm t_{bad} of $\varphi(x^c)$ has already been seen during step i. This then leads to a contradiction as before.

6. THE MAIN RESULT

We are now ready to state the main result of this paper. First, we state the conditions a symbolic protocol should satisfy.

DEFINITION 10. *A CoSP protocol is* randomness-safe *if it satisfies the following conditions:*

1. *The argument of every ek-, dk-, vk-, and sk-computation node and the third argument of every E- and sig-computation node is an N-computation node with $N \in \mathbf{N}_P$. (Here and in the following, we call the nodes referenced by a protocol node its arguments.) We call these N-computation nodes* randomness nodes. *Any two randomness nodes on the same path are annotated with different nonces.*
2. *Every computation node that is the argument of an ek-computation node or of a dk-computation node on some path p occurs only as argument to ek- and dk-computation nodes on that path p.*
3. *Every computation node that is the argument of a vk-computation node or of an sk-computation node on some path p occurs only as argument to vk- and sk-computation nodes on that path p.*
4. *Every computation node that is the third argument of an E-computation node or of a sig-computation node on some path p occurs exactly once as an argument in that path p.*
5. *There are no computation nodes with the constructors garbage, garbageEnc, garbageSig, or $N \in \mathbf{N}_E$.*

In contrast to [4], we do not put any restrictions on the use of keys any more. The requirements above translate to simple syntactic restrictions on the protocols that require us to use each randomness nonce only once. For example, in the applied π-calculus, this would mean that whenever we create a term $enc(e, p, r)$, we require that r is under a restriction νr and used only here.

In addition to randomness-safe protocols, we put a number of conditions on the computational implementation. The cryptographically relevant conditions are PROG-KDM security and malicious-key extractability of the encryption scheme, and strong existential unforgeability of the signature scheme. In addition, we have a large number of additional conditions of syntactic nature, e.g., that the pair-constructor works as expected, that from a ciphertext one can efficiently compute the corresponding encryption key, or that an encryption key uniquely determines its decryption key. These requirements are either natural or can be easily achieved by suitable tagging (e.g., by tagging ciphertexts with their encryption keys). The full list of implementation conditions is given in Appendix A.

THEOREM 1. *The implementation A (satisfying the implementation conditions from Appendix A) is a computationally sound implementation of the symbolic model from Section 2 for the class of randomness-safe protocols. (Note that our definition of computational soundness covers trace properties, not equivalence properties.)*

The full proof of this theorem is given in [7]. From this result, we get, e.g., immediately computational soundness in the applied π-calculus (see [4]) without the restrictions on keys imposed there.

Acknowledgments. Dominique Unruh was supported by the Cluster of Excellence "Multimodal Computing and Interaction", by the European Social Fund's Doctoral Studies and Internationalisation Programme DoRa, by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, by the European Social Fund through the Estonian Doctoral School in Information and Communication Technology. Michael Backes was supported by CISPA (Center for IT-Security, Privacy and Accountability), and by an ERC starting grant. Part of the work was done while Ankit Malik was at MPI-SWS, and while Dominique Unruh was at the Cluster of Excellence "Multimodal Computing and Interaction".

APPENDIX

A. IMPLEMENTATION CONDITIONS

1. A is an implementation of $\mathbf{M}$ in the sense of [4] (in particular, all functions A_f ($f \in \mathbf{C} \cup \mathbf{D}$) are polynomial-time computable).
2. There are disjoint and efficiently recognizable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, signatures, verification keys, signing keys, pairs, and payload-strings. The

set of all bitstrings of type nonce we denote Nonces_k.[13] (Here and in the following, k denotes the security parameter.)

3. The functions A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, and A_{pair} are length-regular. We call an n-ary function f length regular if $|m_i| = |m_i'|$ for $i = 1, \ldots, n$ implies $|f(\underline{m})| = |f(\underline{m'})|$. All $m \in \text{Nonces}_k$ have the same length.
4. A_N for $N \in \mathbf{N}$ returns a uniformly random $r \in \text{Nonces}_k$.
5. Every image of A_{enc} is of type ciphertext, every image of A_{ek} and A_{ekof} is of type encryption key, every image of A_{dk} is of type decryption key, every image of A_{sig} is of type signature, every image of A_{vk} and A_{vkof} is of type verification key, every image of A_{empty}, A_{string_0}, and A_{string_1} is of type payload-string.
6. For all $m_1, m_2 \in \{0,1\}^*$ we have $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of A_{pair}. If m is not of type pair, $A_{fst}(m) = A_{snd}(m) = \bot$.
7. For all m of type payload-string we have that $A_{unstring_i}(A_{string_i}(m)) = m$ and $A_{unstring_i}(A_{string_j}(m)) = \bot$ for $i, j \in \{0,1\}$, $i \neq j$. For $m = \mathit{empty}$ or m not of type payload-string, $A_{unstring_0}(m) = A_{unstring_1}(m) = \bot$. Every m of type payload-string is of the form $m = A_{string_0}(m')$ or $m = A_{string_1}(m')$ or $m = \mathit{empty}$ for some m' of type payload-string. For all m of type payload-string, we have $|A_{string_0}(m)|, |A_{string_1}(m)| > |m|$.
8. $A_{ekof}(A_{enc}(p, x, y)) = p$ for all p of type encryption key, $x \in \{0,1\}^*$, $y \in \text{Nonces}_k$. $A_{ekof}(e) \neq \bot$ for any e of type ciphertext and $A_{ekof}(e) = \bot$ for any e that is not of type ciphertext.
9. $A_{vkof}(A_{sig}(A_{sk}(x), y, z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$, $x, z \in \text{Nonces}_k$. $A_{vkof}(e) \neq \bot$ for any e of type signature and $A_{vkof}(e) = \bot$ for any e that is not of type signature.
10. $A_{enc}(p, m, y) = \bot$ if p is not of type encryption key.
11. $A_{dec}(A_{dk}(r), m) = \bot$ if $r \in \text{Nonces}_k$ and $A_{ekof}(m) \neq A_{ek}(r)$. (This implies that the encryption key is uniquely determined by the decryption key.)
12. $A_{dec}(d, c) = \bot$ if $A_{ekof}(c) \neq A_{ekofdk}(d)$ or $A_{ekofdk}(d) = \bot$.
13. $A_{dec}(d, A_{enc}(A_{ekofdk}(e), m, r)) = m$ if $r \in \text{Nonces}_k$ and $d := A_{ekofdk}(e) \neq \bot$.
14. $A_{ekofdk}(d) = \bot$ if d is not of type decryption key.
15. $A_{ekofdk}(A_{dk}(r)) = A_{ek}(r)$ for all $r \in \text{Nonces}_k$.
16. $A_{vkofsk}(s) = \bot$ if s is not of type signing key.
17. $A_{vkofsk}(A_{sk}(r)) = A_{vk}(r)$ for all $r \in \text{Nonces}_k$.
18. $A_{dec}(A_{dk}(r), A_{enc}(A_{ek}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
19. $A_{verify}(A_{vk}(r), A_{sig}(A_{sk}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
20. For all $p, s \in \{0,1\}^*$ we have that $A_{verify}(p, s) \neq \bot$ implies $A_{vkof}(s) = p$.
21. $A_{isek}(x) = x$ for any x of type encryption key. $A_{isek}(x) = \bot$ for any x not of type encryption key.
22. $A_{isvk}(x) = x$ for any x of type verification key. $A_{isvk}(x) = \bot$ for any x not of type verification key.
23. $A_{isenc}(x) = x$ for any x of type ciphertext. $A_{isenc}(x) = \bot$ for any x not of type ciphertext.
24. $A_{issig}(x) = x$ for any x of type signature. $A_{issig}(x) = \bot$ for any x not of type signature.
25. We define an encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows: KeyGen picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{ek}(r), A_{dk}(r))$. $\mathsf{Enc}(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{enc}(p, m, r)$. $\mathsf{Dec}(k, c)$ returns $A_{dec}(k, c)$. We require that then $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is PROG-KDM secure.
26. Additionally, we require that $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is malicious-key extractable.
27. We define a signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ as follows: $\mathsf{SKeyGen}$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{vk}(r), A_{sk}(r))$. $\mathsf{Sig}(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{sig}(p, m, r)$. $\mathsf{Verify}(p, s, m)$ returns 1 iff $A_{verify}(p, s) = m$. We require that then $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ is strongly existentially unforgeable.
28. For all e of type encryption key and all $m, m' \in \{0,1\}^*$, the probability that $A_{enc}(e, m, r) = A_{enc}(e, m', r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.
29. For all $r_s \in \text{Nonces}_k$ and all $m \in \{0,1\}^*$, the probability that $A_{sig}(A_{sk}(r_s), m, r) = A_{sig}(A_{sk}(r_s), m, r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.
30. A_{ekofdk} is injective. (I.e., the encryption key uniquely determines the decryption key.)
31. A_{vkofsk} is injective. (I.e., the verification key uniquely determines the signing key.)

[13] This would typically be the set of all k-bit strings with a tag denoting nonces.

B. REFERENCES

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[2] M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *LNCS*, pages 3–22. Springer, 2000.

[3] P. Adão, G. Bana, J. Herzog, and A. Scedrov. Soundness and completeness of formal encryption: The cases of key cycles and partial information leakage. *Journal of Computer Security*, 17(5):737–797, 2009.

[4] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *ACM CCS 2009*, pages 66–78, November 2009.

[5] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. IACR Cryptology ePrint Archive 2009/080, 2009. Version from 2009-02-18.

[6] M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *ACM CCS 2010*, pages 387–398. ACM Press, October 2010. Preprint on IACR ePrint 2010/416.

[7] M. Backes, A. Malik, and D. Unruh. Computational Soundness without Protocol Restrictions. IACR ePrint archive, 2012. Full version of this paper.

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

[9] M. Backes, B. Pfitzmann, and A. Scedrov. Key-dependent message security under active attacks - brsim/uc-soundness of dolev-yao-style encryption with key cycles. *Journal of Computer Security*, 16(5):497–530, 2008.

[10] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/2003/015`.

[11] M. Backes and D. Unruh. Computational soundness of symbolic zero-knowledge proofs. *Journal of Computer Security*, 18(6):1077–1155, 2010. Preprint on IACR ePrint 2008/152.

[12] G. Bana and H. Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In P. Degano and J. Guttman, editors, *Principles of Security and Trust*, volume 7215 of *Lecture Notes in Computer Science*, pages 189–208. Springer Berlin / Heidelberg, 2012.

[13] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 2004.

[14] M. Bellare, D. Hofheinz, and S. Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In *EUROCRYPT 2009*, pages 1–35, 2009.

[15] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[16] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology: EUROCRYPT '94*, volume 950 of *LNCS*, pages 92–111. Springer, 1994.

[17] F. Böhl, D. Hofheinz, and D. Kraschewski. On definitions of selective opening security. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 522–539. Springer, 2012.

[18] J. Camenisch, N. Chandran, and V. Shoup. A public key encryption scheme secure against key dependent chosen plaintext and adaptive chosen ciphertext attacks. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *LNCS*, pages 351–368. Springer, 2009.

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

[20] H. Comon-Lundh, V. Cortier, and G. Scerri. Security proof with dishonest keys. In *POST*, pages 149–168, 2012.

[21] 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.

[22] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *Proc. 14th European Symposium on Programming (ESOP)*, pages 157–171, 2005.

[23] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[24] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 34–39, 1983.

[25] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[26] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.

[27] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 25th IEEE Symposium on Security & Privacy*, pages 71–85, 2004.

[28] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[29] L. Mazaré and B. Warinschi. Separating trace mapping and reactive simulatability soundness: The case of adaptive corruption. In P. Degano and L. Viganò, editors, *ARSPA-WITS 2009*, volume 5511 of *LNCS*, pages 193–210. Springer, 2009.

[30] M. Merritt. *Cryptographic Protocols*. PhD thesis, Georgia Institute of Technology, 1983.

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

[32] J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In M. Yung, editor, *Advances in Cryptology, Proceedings of CRYPTO '02*, volume 2442 of *Lecture Notes in Computer Science*, pages 111–126. Springer-Verlag, 2002.

[33] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.

[34] S. Schneider. Security properties and CSP. In *Proc. 17th IEEE Symposium on Security & Privacy*, pages 174–187, 1996.

[35] D. Unruh. Programmable encryption and key-dependent messages. IACR ePrint archive 2012/423, 2012.