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

August 22, 2012

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.

*A short version of this paper appears at ACM CCS 2012 [7].

Contents

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 B 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

[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.

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 appendix

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 matching the following grammar:

$$\begin{aligned} T ::= {} & enc(ek(N), T, N) \mid ek(N) \mid dk(N) \mid \\ & sig(sk(N), T, N) \mid vk(N) \mid sk(N) \mid \\ & pair(T, T) \mid S \mid N \mid \\ & garbage(N) \mid garbageEnc(T, N) \mid \\ & garbageSig(T, N) \\ S ::= {} & empty \mid string_0(S) \mid string_1(S) \end{aligned}$$

where the nonterminal N stands for nonces.

Destructors. $\mathbf{D} := \{dec/2, isenc/1, isek/1, isdk/1, ekof/1, ekofdk/1, verify/2, issig/1, isvk/1, issk/1, vkof/2, vkofsk/1, fst/1, snd/1, unstring_0/1, unstring_1/1, equals/2\}$. The destructors $isek$,

[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.

$$
\begin{aligned}
dec(dk(t_1), enc(ek(t_1), m, t_2)) &= m \\
isenc(enc(ek(t_1), t_2, t_3)) &= enc(ek(t_1), t_2, t_3) \\
isenc(garbageEnc(t_1, t_2)) &= garbageEnc(t_1, t_2) \\
isek(ek(t)) &= ek(t) \\
isdk(dk(t)) &= dk(t) \\
ekof(enc(ek(t_1), m, t_2)) &= ek(t_1) \\
ekof(garbageEnc(t_1, t_2)) &= t_1 \\
ekofdk(dk(t_1)) &= ek(t_1) \\
\\
verify(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 \\
issig(sig(sk(t_1), t_2, t_3)) &= sig(sk(t_1), t_2, t_3) \\
issig(garbageSig(t_1, t_2)) &= garbageSig(t_1, t_2) \\
isvk(vk(t_1)) &= vk(t_1) \\
issk(sk(t)) &= sk(t) \\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) \\
vkof(garbageSig(t_1, t_2)) &= t_1 \\
vkofsk(sk(t_1)) &= vk(t_1) \\
\\
fst(pair(x, y)) &= x \\
snd(pair(x, y)) &= y \\
unstring_0(string_0(s)) &= s \\
unstring_1(string_1(s)) &= s \\
equals(t_1, t_1) &= t_1
\end{aligned}
$$

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

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. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ 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 $unstring_0$ and $unstring_1$ allow to parse payload-strings. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, $unstring_i$, and $equals(\cdot, empty)$.)

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. $\text{eval}_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ if $f(t_1, \ldots, t_n) \neq \bot$ and $f(t_1, \ldots, t_n) \in \mathbf{T}$ and $\text{eval}_f(t_1, \ldots, t_n) = \bot$ otherwise.

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

Figure 2: Deduction rules for the symbolic model

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 non-deterministically 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.

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 (Computational soundness – simplified [4]) *We say a computational implementation A is a computationally sound implementation of a symbolic model for a 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.*

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

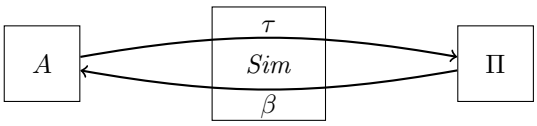

Figure 3: A typical CoSP simulator

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 is also 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 in Appendix C.)

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 $\mathcal{O}^N_{enc}$. 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 $t := enc(\ldots)$, 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. 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.)

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 $enc(\ldots, N)$ 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 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

[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.

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 $reg_R := m$.*
- $R := \mathtt{enc}_{\mathtt{ch}}(N, R_1)$*: 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 $reg_{R_i} \neq \bot$.

[5]Note that from the description of a circuit, it is possible to determine the length of its output. This will be important in the definition of `FCLen` below.

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 $\perp$). 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 $reg_R = \perp$). Registers reg_N are either undefined ($reg_N = \perp$), 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:*

[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.

- $R := \texttt{getek}_{\texttt{ch}}(N)$ *or* $R := \texttt{getdk}_{\texttt{ch}}(N)$ *or* $R := \texttt{eval}_{\texttt{ch}}(C, R_1, \ldots, R_n)$ *or* $R := \texttt{enc}_{\texttt{ch}}(N, R_1)$*: Set* $reg_R :=$ "$\texttt{getek}_{\texttt{ch}}(N)$" *or* $reg_R :=$ "$\texttt{getdk}_{\texttt{ch}}(N)$" *or* $reg_R :=$ "$\texttt{eval}_{\texttt{ch}}(C, R_1, \ldots, R_n)$" *or* $reg_R :=$ "$\texttt{enc}_{\texttt{ch}}(N, R_1)$"*, respectively.*
- $\texttt{dec}_{\texttt{ch}}(N, c)$*: If* $c \in cipher_N$*, return* `forbidden`*. Otherwise, query* $\texttt{dec}_{\texttt{cs}}(c)$ *from* CS_N *and return its response.*
- $\texttt{oracle}_{\texttt{ch}}(x)$*: Return* $\mathcal{O}(x)$*.*
- $\texttt{reveal}_{\texttt{ch}}(R_1)$*: Compute* $m \leftarrow \texttt{FCRetrieve}(R_1)$*. (*`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* $reg_R =$ "$\texttt{getek}_{\texttt{ch}}(N)$"*: Query* CS_N *with* $\texttt{getek}_{\texttt{cs}}()$*. Store the answer in* reg_R*. Return* reg_R*.*
- *If* $reg_R =$ "$\texttt{eval}_{\texttt{ch}}(C, R_1, \ldots, R_n)$"*: Compute* $m_i := \texttt{FCRetrieve}(R_i)$ *for* $i = 1, \ldots, n$*. Compute* $m' := C(m_1, \ldots, m_n)$*. Set* $reg_R := m'$*. Return* m'*.*
- *If* $reg_R =$ "$\texttt{enc}_{\texttt{ch}}(N, R_1)$" *and there was no* $\texttt{getdk}_{\texttt{cs}}()$*-query to* CS_N *yet: Compute* $l := \texttt{FCLen}(R_1)$*. (*`FCLen` *is defined in Definition 7 below.) Query* CS_N *with* $\texttt{fakeenc}_{\texttt{cs}}(R, l)$*. 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* $reg_R =$ "$\texttt{enc}_{\texttt{ch}}(N, R_1)$" *and there was a* $\texttt{getdk}_{\texttt{cs}}()$*-query to* CS_N*: Compute* $m := \texttt{FCRetrieve}(R_1)$*. Query* CS_N *with* $\texttt{enc}_{\texttt{cs}}(R, m)$*. 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* $reg_R =$ "$\texttt{getdk}_{\texttt{ch}}(N)$"*: Query* CS_N *with* $\texttt{getdk}_{\texttt{cs}}()$*. Store the answer in* reg_R*. If this was the first* $\texttt{getdk}_{\texttt{cs}}(N)$*-query for that value of* N*, do the following 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* $reg_R \neq \bot$*) 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* $reg_R =$ "$\mathtt{enc}_{\mathtt{ch}}(N, R_1)$"*: Let* ℓ_c *be as in Definition 6. Return* $\ell_c(\eta, \mathtt{FCLen}(R_1))$.

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 $R := \mathtt{eval}_{\mathtt{ch}}(C, \dots)$). 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 $i = 0, \dots, n/2$ and $j = 0, \dots, n$.

[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 $R_i := \mathtt{enc}_{\mathtt{ch}}(N, R_{i-1})$ 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 $e_I := \mathsf{Enc}(dk_{i_1}, \dots \mathsf{Enc}(dk_{i_{n/2}}, s) \dots)$ for each set $I = \{i_1, \dots, i_{n/2}$ of size $n/2$. But that protocol would need to send an exponential number of ciphertexts I.

- 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 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

[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.

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 destructors), 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.)

The need for malicious-key extractability. 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 produces 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 $S\varphi \nvdash t\varphi$ 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

[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.

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 are given in Appendix B.

Theorem 1 *The implementation A (satisfying the implementation conditions from Appendix B) 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 Appendix C. From this result, we get, e.g., immediately computational soundness in the applied π-calculus (see [4]) without the restrictions on keys imposed there.

7 Proof sketch

We now present a proof sketch of Theorem 1. We have highlighted the changes with respect of the proof sketch to the original CoSP result (Section 4) in blue. There is a certain amount of redundancy with Section 4 since we tried to make this section self-contained. The full proof is presented in Appendix C.

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.) In contrast to the situation described in Section 4, we allow the simulator to produce incomplete terms. These may contain variables x^m, standing for subterms the simulator has not figured out yet. Whenever the protocol makes a decision that depends on the as yet undetermined values of these variables (e.g., when branching depends on the applicability of a destructor which in turn depends on the value to be assigned to x^m), the simulator is asked what the correct decision would be (i.e., the simulator is asked whether the destructor application would succeed).

The simulator has to satisfy the following three 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.
- Consistency: The simulator outputs an assignment φ to all variables x^c in the end of the execution. This assignment must guarantee that any decision the simulator made for the protocol was correct. I.e., when the simulator said a destructor application D (containing variables x^c) succeeds, then $D\varphi$ must actually succeed. And vice versa.
- Abort-freeness: The simulator does not abort. Since our simulator will not have an abort instruction, this property will be automatically fulfilled.

The existence of such a simulator (for any A) then guarantees computational soundness: Dolev-Yaoness together with consistency 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 the 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 is also 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* not to introduce contradictory mappings (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 $\tau(c)$.

However, what do we do if we have to parse a ciphertext c that we cannot decrypt? In the original CoSP proof (where secret keys are never sent), we could safely parse $\tau(c) := garbageEnc(ek(N^e), N^c)$ for suitable nonces N^e, N^c; as the decryption key is never revealed, we never notice if $\tau(c)$ is actually a valid encryption. But this approach leads to problems in out setting when the decryption key is later revealed. Then we suddenly notice that $\tau(c)$ should be $enc(ek(N^e), m, N^c)$ for some plaintext m. We avoid this problem by not deciding right away whether $\tau(c)$ should be $garbageEnc(\dots)$ or $enc(\dots)$. The simulator just returns $\tau(c) := x^c$, and only at end of the execution, he assigns $\varphi(x^c) := enc(\dots)$ if he has learned the decryption key by then, and $\varphi(x^c) := garbageEnc(\dots)$ otherwise. (And we extend the definition of β to translate $\beta(x^c) = c$ as expected.)

It remains to clarify how the simulator answers questions. I.e., given a destructor term D, how does the simulator decide whether an evaluation of $D\varphi$ succeeds or not (where φ maps each x^c to the $garbageEnc(\dots)$ or $enc(\dots)$, depending on information the simulator does not have yet). It turns out that in most situations, whether a destructor application succeeds or not does not depend on whether a particular x^c is assigned $garbageEnc(\dots)$ or $enc(\dots)$. The only case where this information would be needed is in an application $dec(sk(N), x^c)$, which will only work if x^c is assigned $enc(ek(N), \dots)$. Fortunately, this case only arises when $sk(N)$ occurs. This in turn only happens when the simulator has already seen the decryption key needed for decrypting c. And given that decryption key, the simulator can figure out whether x^c will be assigned a term of the form $enc(\dots)$ and what its plaintext is.

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 $m := A_{dec}(A_{dk}(r_M), c) \neq \bot$
- $\tau(c) := x^c$ if c is of type ciphertext
- $\beta(enc(ek(N), t, M)) := A_{enc}(A_{ek}(r_N), \beta(t), r_M)$ if $M \in \mathbf{N}_P$
- $\beta(enc(ek(M), t, N^m)) := m$ if $M \in \mathbf{N}_P$
- $\beta(x^c) := c$

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 $\beta(F(t_1, \ldots, t_n)) = A_F(\beta(t_1), \ldots, \beta(t_n))$. (The precise formulation of the invariant is somewhat more complex, because the actual terms t_i are not known during the execution. We only know terms t_i^* that are partially evaluated and still contain destructors and variables. To deal with this, we define $t_i := \mathit{red}(t_i^*)$ where red is a suitable reduction algorithm that simplifies the t_i and removes all destructors and some variables.) 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. (Our proof is similar to the one in the original CoSP setting, except that we have a number of additional cases to check and have to deal with the reduction algorithm red in many places.)

Showing consistency. The proof of the consistency of the simulator consists mainly of checking that in all cases, the reduction algorithm red returns values compatible with those that will be assigned to the variables x^c in the end and that thus all answers given by the simulator are those that would be given if the simulator knew these assignments earlier on.

Showing Dolev-Yaoness. The proof of Dolev-Yaoness is where most of the actual cryptographic assumptions come in. Starting from the simulator Sim, we introduce a sequence of simulators Sim_2, Sim_3, Sim_4, Sim_5, Sim_7. (We have gaps in the numbering because in this overview we omit the simulators Sim_1 and Sim_6 which only serve minor technical purposes.)

In Sim_2, we maintain an instance of the real challenger (see Definition 2), and we change the function β as follows: When invoked as $\beta(\mathit{enc}(\mathit{ek}(N), t, M))$ with $M \in \mathbf{N}_P$, instead of computing $A_{enc}(A_{ek}(r_N), \beta(t), r_M)$, β uses the real challenger to produce the ciphertext c. More precisely, $\beta(\mathit{enc}(\mathit{ek}(N), t, M))$ sends a sequence of $\mathtt{getek}_{\mathtt{ch}}$-, $\mathtt{getdk}_{\mathtt{ch}}$-, $\mathtt{eval}_{\mathtt{ch}}$-, and $\mathtt{dec}_{\mathtt{ch}}$-queries to the real challenger that have the effect that the real challenger internally computes $m := \beta(t)$ and stores the result in some register reg_R. Then β issues an $\mathtt{enc}_{\mathtt{ch}}(N, R)$-query to compute an encryption c of m and a $\mathtt{reveal}_{\mathtt{ch}}$-query to reveal c. Finally, β returns c. Similarly, $\beta(\mathit{ek}(N))$ returns the public key provided by the real challenger. Note that this construction makes sure that the simulator will not see the intermediate values from the computation of the plaintext of c (they stay inside the registers of the real challenger). This is important since we will have to argue later that the simulator cannot guess any of the plaintexts that that the adversary would not know in the symbolic setting. The function τ is changed to issue a $\mathtt{dec}_{\mathtt{ch}}$-query whenever it needs to decrypt a ciphertext while parsing. Notice that if c was returned by $\beta(t)$ with $t := \mathit{enc}(\ldots)$, then $\tau(c)$ just recalls the term t without having to decrypt. Hence the real challenger is never asked to decrypt a ciphertext it produced. 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 different places.)

To show that the simulator is Dolev-Yao we have to show that whenever the simulator sends a term $t = \tau(m)$ to the protocol, then $S\varphi \vdash t\varphi$ where S are the terms received so far, and φ assigns terms to variables x^c. We do this by showing that if $t = \tau(m)$ does not satisfy this condition, then there is some subterm t_{bad} of $t\varphi$ that would not have been output by τ. But for this reasoning it is necessary that every subterm of $t\varphi$ has already been computed at the time when t is sent. This is not the case since t_{bad} might actually be a subterm of $\varphi(x^c)$ for some x^c occurring in t. And those subterms may not be computed when $t = \tau(m)$ is invoked, because to compute them, we would need to know the plaintext m' of c which may not be known yet. In order to make sure that $\tau(m')$ is computed, we use the MKE property of the encryption

scheme (see Definition 9). This property allows us to construct a simulator Sim_3 that, for every ciphertext it encounters, computes all candidate plaintexts m', and invokes $\tau(m')$ for each. This will make sure that later, whenever t_{bad} is a subterm of the term $t = \tau(m)$ sent in a certain iteration, then $t_{bad} = \tau(m_{bad})$ has been computed in that iteration.

In Sim_4, we replace the real challenger by the fake challenger. Since the real challenger is never asked to decrypt a ciphertext it produced, PROG-KDM security guarantees that the hybrid executions of Sim_3 and Sim_4 are indistinguishable.

In the original CoSP proof, at this point we argued that, since the fake encryption oracle encrypts 0-plaintexts, we can remove the recursive computation of $\beta(t)$ in an invocation $\beta(enc(N, t, M))$. This was needed to show that secrets contained in plaintexts are never accessed. In our setting, the argumentation will be more complex, since still use the true plaintext $\beta(t)$, only we outsourced the computation of $\beta(t)$ for a plaintext t to the fake challenger (see the construction of Sim_2). The next simulator will take care of this.

We now change the simulator Sim_4 into a simulator Sim_5 that calls the ciphertext simulators directly. Essentially, we make the definition of the fake challenger explicit. Remember that the fake challenger lazily computes necessary plaintexts only when needed for opening a ciphertext. When a ciphertext's decryption key is not revealed, due to the programmability of the encryption scheme (the ciphertext simulator provides that programmability), the corresponding plaintext is not needed and thus never computed.

Now for the simulator Sim_5 we can show that whenever $\beta(t)$ is called for some term t, then $S\varphi \vdash t\varphi$. In consequence, $\beta(t)$ will never access any values that would, symbolically, be secret. (E.g., if $S\varphi \nvdash N$, then no $\beta(t)$-invocation will access the computational value r_N of the nonce N.)

Finally, in Sim_7, we additionally change β to use a signing oracle in order to produce signatures. Analogous to Sim and Sim_2, the hybrid executions of Sim_5 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.

As described in the construction of Sim_5, whenever Sim_7 invokes $\beta(t)$, then $S\varphi \vdash t\varphi$ holds.

We prove that whenever $S\varphi \nvdash t\varphi$, then $t\varphi$ contains a visible subterm t_{bad} with $S\varphi \nvdash t_{bad}$ such that t_{bad} is a protocol nonce, or a ciphertext $enc(\ldots, N)$ 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 $t\varphi$ has a visible subterm t_{bad} satisfying a number of conditions. Visibility implies that the recursive computation of $\tau(m)$ had a subinvocation $\tau(m_{bad}) = t_{bad}$. (The use of the MKE property in Sim_3 ensures that.) 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.

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, and by grant ETF9171 from the Estonian Science Foundation. 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".

A Symbolic model

In Sections A–C we describe the full details of our result. Changes (beyond simple presentation matters) with respect to the proof from [4] are highlighted in blue.

We first specify the symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$:

- 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. 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: We define $\mathbf{T}$ as the set of all terms T matching the following grammar:

$$\begin{aligned} T ::= {} & enc(ek(N), T, N) \mid ek(N) \mid dk(N) \mid \\ & sig(sk(N), T, N) \mid vk(N) \mid sk(N) \mid \\ & pair(T, T) \mid S \mid N \mid \\ & garbage(N) \mid garbageEnc(T, N) \mid \\ & garbageSig(T, N) \\ S ::= {} & empty \mid string_0(S) \mid string_1(S) \end{aligned}$$

  where the nonterminal N stands for nonces.
- Destructors: $\mathbf{D} := \{dec/2, isenc/1, isek/1, isdk/1, ekof/1, ekofdk/1, verify/2, issig/1, isvk/1, issk/1, vkof/2, vkofsk/1, fst/1, snd/1, unstring_0/1, unstring_1/1, equals/2\}$. 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. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ 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 $unstring_0$ and $unstring_1$ allow to parse payload-strings. (Destructors $ispair$ and $isstring$ are not necessary, they can be emulated using fst, $unstring_i$, and $equals(\cdot, empty)$.) The behavior of the destructors is given 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.

B Computational implementation

The computational implementation. Obtaining a computational soundness result for the symbolic model $\mathbf{M}$ requires its implementation to use an PROG-KDM secure encryption scheme and a strongly existentially unforgeable signature scheme. More precisely, we require that (A_{ek}, A_{dk}), A_{enc}, and A_{dec} form the key generation, encryption and decryption algorithm of an PROG-KDM-secure scheme; and that (A_{vk}, A_{sk}), A_{sig}, and A_{verify} form the key generation, signing, and verification algorithm of a strongly existentially unforgeable signature scheme. Let $A_{isenc}(m) = m$ iff m is a ciphertext. (Only a syntactic check is performed; it is not necessary to check whether m was correctly generated.) A_{issig}, A_{isek}, and A_{isvk} are defined analogously. A_{ekof} extracts the encryption key from a ciphertext, i.e., we assume that ciphertexts are tagged with their encryption key. Similarly A_{vkof} extracts the verification key from a signature, and A_{verify} can be used to extract the signed message from a signature, i.e., we assume that signatures are tagged with their verification key and the signed message. Nonces are implemented as (suitably tagged) random k-bit strings. A_{pair}, A_{fst}, and A_{snd} construct and destruct pairs. We require that the implementation of the constructors are length regular, i.e., the length of the result of applying a constructor depends only on the lengths of the arguments. No restrictions are put on $A_{garbage}$, $A_{garbageEnc}$, and $A_{garbageSig}$ as these are never actually used by the protocol. (The implementation of these functions need not even fulfill equations like $A_{isenc}(A_{garbageEnc}(x)) = A_{garbageEnc}(x)$.)

The exact requirements are as follows:

Implementation conditions. We require that the implementation A of the symbolic model $\mathbf{M}$ has the following properties:

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 $m = empty$ 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 $m = empty$ 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

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

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.)

C Computational soundness proof

C.1 Construction of the simulator

For the construction of the simulator, we will consider terms containing variables. We use an infinite set of variables x^c where c ranges over all c of type bitstring. In analogy to the message

type $\mathbf{T}$, we define the set $\mathbf{T}_x$ to consist of all terms T matching the following grammar:

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

Note that the only difference to the grammar defining $\mathbf{T}$ is that in the grammar for $\mathbf{T}_x$, we have a production rule $T ::= x^c$ where x^c ranges over all variables. We additionally define the set $T_{\mathbf{D},x}$ of "destructor terms", i.e., of all terms built from constructors, destructors, nonces, and variables.

Additionally, we extend the partial functions that represent the destructors (see Figure 1) to $\mathbf{T}_x$. A destructor $D/n \in \mathbf{D}$ is a then a partial function $D : \mathbf{T}_x^n \to \mathbf{T}_x$ described by the rules from Figure 1, and the following two new rules:

$$\begin{aligned} \mathit{isenc}(x^c) &= x^c \\ \mathit{ekof}(x^c) &= \mathit{ek}(N^{A_{\mathit{ekof}}(c)}) \end{aligned}$$

The rules for *ekof* may seem strange because it means that the behavior of the destructor (part of the symbolic model) depends on the computational implementation A_{ekof}. We stress that these non-standard definitions are only a tool used in the construction of the simulator and in the proof. The reason for this rule is that x^c essentially stands for a (still unknown) term of the form $\mathit{enc}(\mathit{ek}(N^{A_{\mathit{ekof}}(c)}, \dots))$ or $\mathit{garbageEnc}(\mathit{ek}(N^{eA_{\mathit{ekof}}(c)}), \dots)$.

We extend the definition of eval_f (see page 5) to $\mathbf{T}_x$, too: For $f/n \in \mathbf{D}$ and $t_1, \dots, t_n \in \mathbf{T}_x$, $\mathrm{eval}_f(t_1, \dots, t_n) := f(t_1, \dots, t_n)$ where the partial function f is as defined above. For $f/n \in \mathbf{C}$ and $t_1, \dots, t_n \in \mathbf{T}_x$, $\mathrm{eval}_f(t_1, \dots, t_n) := f(t_1, \dots, t_n)$ if $f(t_1, \dots, t_n) \in \mathbf{T}_x$, and $\mathrm{eval}_f(t_1, \dots, t_n) = \bot$ otherwise.

Finally, define the function eval as follows: $\mathrm{eval}(N) = N$ for $N \in \mathbf{N}$, $\mathrm{eval}(x^c) = x^c$, $\mathrm{eval}(f(t_1, \dots, t_n)) = \mathrm{eval}_f(\mathrm{eval}(t_1), \dots, \mathrm{eval}(t_n))$ where eval_f is as defined above.

Notice that $\mathbf{T}_x$, restricted to terms without variables, is $\mathbf{T}$. And when restricted to $\mathbf{T}$, the destructor functions and eval_f coincide with their original definitions. Thus we can use the same symbols for the old and the new definitions without ambiguity.

Construction of the simulator. In the following, we define distinct nonces $N^m \in \mathbf{N}_E$ for each $m \in \{0,1\}^*$. In a hybrid execution, we call a term t *honestly generated* if it occurs as a subterm of a term sent by the protocol Π^C to the simulator before it has occurred as a subterm of a term sent by the simulator to the protocol Π^C.

For an adversary E and a polynomial p, we construct the simulator *Sim* as follows: In the first activation, it chooses $r_N \in \mathrm{Nonces}_k$ for every $N \in \mathbf{N}_P$. It maintains an integer *len*, initially 0. At any point in the execution, $\mathcal{N}$ denotes the set of all nonces $N \in \mathbf{N}_P$ that occurred in terms received from Π^C. $\mathcal{R}$ denotes the set of *randomness nonces* (i.e., the nonces associated with all randomness nodes of Π^C passed through up to that point).

Sim internally simulates the adversary E. When receiving a destructor term $\tilde{t} \in T_{\mathbf{D},x}$ from Π^C in the i-th step, it passes $\beta(\mathit{red}_i(\tilde{t}))$ to E where the partial function $\beta : \mathbf{T}_x \to \{0,1\}^*$ and the reduction function red_i are defined below. When E answers with $m \in \{0,1\}^*$, the simulator sends $\tau(m)$ to Π^C where the function $\tau : \{0,1\}^* \to \mathbf{T}_x$ is defined below. The bitstrings sent from the protocol at control nodes are passed through to E and vice versa. When the simulator

receives $(info, \nu, t)$, the simulator increases len by $\ell(t)+1$ where $\ell : \mathbf{T} \to \{0,1\}^*$ is defined below. If $len > p(k)$, the simulator terminates, otherwise it answers with $(proceed)$.

Normalizing destructor terms. We define the partial function $red_i : T_{\mathbf{D},x} \to \mathbf{T}_x$ recursively as follows:

- For a nonce N: $red_i(N) := N$.
- For a constructor or destructor f: $red_i(f(t_1, \ldots, t_n)) := \mathrm{eval}_f(red_i(t_1), \ldots, red_i(t_n))$.
- For any c such that $dk(N^e)$ with $e := A_{ekof}(c)$ occurred until step i of the execution: $red_i(x^c) := enc(\tau(e), red_i(\tau(A_{dec}(A^{-1}_{ekofdk}(e), c))), N^c)$ if $A_{dec}(A^{-1}_{ekofdk}(e), c) \neq \bot$ and $red_i(x^c) := garbageEnc(\tau(e), N^c)$ if $A_{dec}(A^{-1}_{ekofdk}(e), c) = \bot$.
- For any c such that $dk(N^e)$ with $e := A_{ekof}(c)$ did not occur until step i of the execution: $red_i(x^c) := x^c$.

Translation functions. The partial function $\beta : \mathbf{T} \to \{0,1\}^*$ is defined as follows (where the first matching rule is taken):

- $\beta(N) := r_N$ if $N \in \mathcal{N}$.
- $\beta(N^m) := m$.
- $\beta(enc(ek(t_1), t_2, M)) := A_{enc}(\beta(ek(t_1)), \beta(t_2), r_M)$ if $M \in \mathcal{R}$.
- $\beta(enc(ek(t_1), t, N^m)) := m$.
- $\beta(x^c) := c$.
- $\beta(ek(N)) := A_{ek}(r_N)$ if $N \in \mathcal{R}$.
- $\beta(ek(N^m)) := m$.
- $\beta(dk(N)) := A_{dk}(r_N)$ if $N \in \mathcal{R}$. Before returning the value $\beta(dk(N))$ invokes $\beta(ek(N))$ and discards it return value. (This is to guarantee that $A_{ek}(N)$ can only be guessed when $\beta(ek(N))$ was invoked.)
- $\beta(dk(N^m)) := A^{-1}_{ekofdk}(m)$. (Note that due to implementation condition 30, there is at most one value $A^{-1}_{ekofdk}(m)$. And see below for a discussion of the polynomial-time computability of $A^{-1}_{ekofdk}(m)$.)
- $\beta(sig(sk(N), t, M)) := A_{sig}(A_{sk}(r_N), \beta(t), r_M)$ if $N, M \in \mathcal{R}$.
- $\beta(sig(sk(N^v), t, M)) := A_{sig}(A^{-1}_{vkofsk}(v), \beta(t), r_M)$ if $M \in \mathcal{R}$. (Note that due to implementation condition 31, there is at most one value $A^{-1}_{vkofsk}(m)$. And see below for a discussion of the polynomial-time computability of $A^{-1}_{vkofsk}(m)$.)
- $\beta(sig(sk(t_1), t, N^s)) := s$.
- $\beta(vk(N)) := A_{vk}(r_N)$ if $N \in \mathcal{R}$.
- $\beta(vk(N^m)) := m$.
- $\beta(sk(N)) := A_{sk}(r_N)$ if $N \in \mathcal{R}$. Before returning the value $\beta(sk(N))$ invokes $\beta(vk(N))$ and discards it return value. (This is to guarantee that $A_{vk}(N)$ can only be guess when $\beta(vk(N))$ was invoked.)
- $\beta(sk(N^m)) := A^{-1}_{vkofsk}(m)$. (Note that due to implementation condition 31, there is at most one value $A^{-1}_{vkofsk}(m)$.)
- $\beta(pair(t_1, t_2)) := A_{pair}(\beta(t_1), \beta(t_2))$.
- $\beta(string_0(t)) := A_{string_0}(\beta(t))$.
- $\beta(string_1(t)) := A_{string_1}(\beta(t))$.
- $\beta(empty) := A_{empty}()$.
- $\beta(garbage(N^c)) := c$.
- $\beta(garbageEnc(t, N^c)) := c$.
- $\beta(garbageSig(t_1, t_2, N^s)) := s$.
- $\beta(t) := \bot$ in all other cases.

The total function $\tau : \{0,1\}^* \to \mathbf{T}_x$ is defined as follows (where the first matching rule is taken):

- $\tau(r) := N$ if $r = r_N$ for some $N \in \mathcal{N} \setminus \mathcal{R}$.
- $\tau(r) := N^r$ if r is of type nonce.
- $\tau(c) := enc(ek(M), t, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}$, $N \in \mathcal{R}$.
- $\tau(c) := enc(ek(N), \tau(m), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ and $m := A_{dec}(A_{dk}(r_N), c) \neq \bot$.
- $\tau(c) := garbageEnc(ek(N), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ but $A_{dec}(A_{dk}(r_N), c) = \bot$.
- $\tau(c) := x^c$ if c is of type ciphertext but $\tau(A_{ekof}(c)) \neq ek(N)$ for all $N \in \mathcal{R}$.
- $\tau(e) := ek(N)$ if e has earlier been output by $\beta(ek(N))$ for some $N \in \mathcal{R}$.
- $\tau(e) := ek(N^e)$ if e is of type encryption key.
- $\tau(k) := dk(N)$ if k has earlier been output by $\beta(dk(N))$ for some $N \in \mathcal{R}$.
- $\tau(k) := dk(N^e)$ if k is of type decryption key and $e := A_{ekofdk}(k) \neq \bot$.
- $\tau(s) := sig(sk(M), t, N)$ if s has earlier been output by $\beta(sig(sk(M), t, N))$ for some $M \in \mathbf{N}$ and $N \in \mathcal{R}$.
- $\tau(s) := sig(sk(M), \tau(m), N^s)$ if s is of type signature and $\tau(A_{vkof}(s)) = vk(M)$ for some $M \in \mathbf{N}$ and $m := A_{verify}(A_{vkof}(s), s) \neq \bot$.
- $\tau(e) := vk(N)$ if e has earlier been output by $\beta(vk(N))$ for some $N \in \mathcal{R}$.
- $\tau(e) := vk(N^e)$ if e is of type verification key.
- $\tau(k) := sk(N)$ if k has earlier been output by $\beta(sk(N))$ for some $N \in \mathcal{R}$.
- $\tau(k) := sk(N^v)$ if k is of type decryption key and $v := A_{vkofsk}(k) \neq \bot$.
- $\tau(m) := pair(\tau(A_{fst}(m)), \tau(A_{snd}(m)))$ if m of type pair.
- $\tau(m) := string_0(m')$ if m is of type payload-string and $m' := A_{unstring_0}(m) \neq \bot$.
- $\tau(m) := string_1(m')$ if m is of type payload-string and $m' := A_{unstring_1}(m) \neq \bot$.
- $\tau(m) := empty$ if m is of type payload-string and $m = A_{empty}()$.
- $\tau(s) := garbageSig(\tau(A_{vkof}(s)), N^s)$ if s is of type signature.
- $\tau(m) := garbage(N^m)$ otherwise.

Note that the recursive definition of τ is well-defined (i.e., terminates): $\tau(c)$ for a ciphertext c recurses into $\tau(m)$ where $m = A_{dec}(\ldots, c)$, and $\tau(s)$ for a signature s recurses into $\tau(m)$ where $m = A_{verify}(\ldots, s)$, and $\tau(m)$ recurses into $\tau(m_1), \tau(m_2)$ when $m = A_{pair}(m_1, m_2)$. In all cases, m/m_i will be shorter than $c/s/m$. (Because c and s additionally carry the information about the public key, and encryption/signing is length regular.) Thus the recursion terminates.

The function $\ell : \mathbf{T} \to \{0,1\}^*$ is defined as $\ell(t) := |\beta(t)|$. Note that $\ell(t)$ does not depend on the actual values of r_N because of the length-regularity of A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, A_{pair}, A_{string_0}, and A_{string_1}. Hence $\ell(t)$ can be computed without accessing r_N.

The final substitution. To define the final substitution, let n be the last step of the execution of the simulator. Let $red := red_n$, and let X be the set of all variables x^c that occurred in the output of τ during the execution. Let σ be the substitution that maps any x^c with c of type ciphertext to the term $garbageEnc(ek(N^e), N^c)$. We define the final substitution $\varphi : X \to \mathbf{T}$ by $\varphi(x^c) := red_n(x^c)\sigma$.

Answering questions. Given a question $Q \in T_{\mathbf{D},x}$ in step i, the simulator computes $red_i(Q)$. If $red_i(Q) = \bot$, the simulator answers no, otherwise he answers yes.

Polynomial-time implementation of the simulator. Notice that in the decryption of the simulator, we apply the functions A^{-1}_{ekofdk} and A^{-1}_{vkofsk}. These functions are not efficiently computable. Nevertheless, the simulator can be implemented in polynomial-time. Notice that a term $dk(N^e)$ is only produced by $\tau(d)$ with $e = A_{ekofdk}(d)$. Thus, whenever $dk(N^e)$ occurs, the simulator has seen the value $d = A^{-1}_{ekofdk}(e)$. Thus, by keeping a record of all these values d, the

simulator can efficiently compute $A^{-1}_{ekofdk}(e)$ whenever needed. Analogously for A^{-1}_{vkofsk}.

C.2 The faking simulators

We define a sequence of simulators $Sim_1, \ldots, Sim_7$. Each will be indistinguishable of the preceding one (this is shown in Lemma 2 below).

Using fresh randomness for encryption. The first simulator Sim_1 is defined like Sim, except that we change the definition of β as follows: When invoking $\beta(ek(N))$ or $\beta(dk(N))$, a fresh key pair (ek_N, dk_N) is chosen as $(ek_N, dk_N) \leftarrow \mathsf{KeyGen}()$ (KeyGen is defined in implementation condition 25), unless this key pair has been chosen before, and ek_N or dk_N is returned, respectively. Similarly, in $\tau(c)$, instead of invoking $A_{dec}(A_{dk}(r_N), c)$, we invoke $A_{dec}(dk_N, c)$. (That is, instead of using the randomness r_N, we use fresh randomness for the key generation.) When invoking $\beta(enc(ek(t_1), t_2, M))$, instead of computing $A_{enc}(\beta(ek(t_1)), \beta(t_2), r_M)$, we compute $\mathsf{Enc}(\beta(ek(t_1)), \beta(t_2))$ (Enc is defined in implementation condition 25).

Using the real challenger. The simulator Sim_2 is defined like Sim_1, except for the following changes:

- Sim_2 maintains an instance of the real challenger (Definition 2) for the encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ (defined in implementation condition 25).
- Instead of invoking $A_{dec}(dk_N, c)$ in a call to $\tau(c)$, we query the real challenger with $\mathtt{dec}_{\mathtt{ch}}(N, c)$. The bitstring returned by $\mathtt{dec}_{\mathtt{ch}}(N, c)$ is used instead of $A_{dec}(dk_N, c)$.
- When the adversary E queries the random oracle on input x, Sim_2 queries $\mathtt{oracle}_{\mathtt{ch}}(x)$ from the real challenger instead.
- Instead of computing $\beta(t)$ to produce a bitstring to be sent to the adversary, the simulator runs $R := \beta^*(t)$ (see below), queries $\mathtt{reveal}_{\mathtt{ch}}(R)$ from the real challenger, and uses the output of the $\mathtt{reveal}_{\mathtt{ch}}(R)$ query instead of $\beta(t)$. The procedure $\beta^*(t)$ is defined as follows:
  - If there already was a call $\beta^*(t)$ with the same t, the register R returned by that call $\beta^*(t)$ is returned. (In particular, we do not repeat the queries performed by the first $\beta^*(t)$-call.)
  - $\beta^*(ek(N))$ picks a fresh register name R, queries $R := \mathtt{getek}_{\mathtt{ch}}(N)$ from the real challenger, and returns R.
  - $\beta^*(dk(N))$ invokes $\beta^*(ek(N))$ (discarding the output), picks a fresh register name R, queries $R := \mathtt{getek}_{\mathtt{ch}}(N)$ from the real challenger, and returns R.
  - $\beta^*(enc(ek(N), t_2, M))$ with $N, M \in \mathcal{R}$ runs $R := \beta^*(t_2)$, picks a fresh register name R', queries $R' := \mathtt{enc}_{\mathtt{ch}}(N, R)$, and returns R'.
  - $\beta^*(enc(ek(N), t_2, M))$ with $N \in \mathbf{N}_E$, $M \in \mathcal{R}$ picks some randomness r, computes the Boolean circuit C that on input e, m with lengths $\ell(ek(M))$ and $\ell(t_2)$ computes $\mathsf{Enc}(e, m)$ with randomness r (Enc is defined in implementation condition 25). Then β^* queries $R_1 := \beta^*(ek(N))$, $R_2 := \beta^*(t_2)$, picks a fresh register name R, queries $R := \mathtt{eval}_{\mathtt{ch}}(C, R_1, R_2)$, and returns R.
  - For all other terms, β^* behaves analogously to β, except that it computes the value that β would compute within a register. More precisely: If $\beta(t)$ performs calls $m_1 := \beta(t_1)$, $\ldots$, $m_n := \beta(t_n)$ and then returns $f(m_1, \ldots, m_n)$ for some function f, then $\beta^*(t)$ invokes $R_1 := \beta^*(t_1)$, $\ldots$, $R_n := \beta^*(t_n)$, computes the circuit C_f that evaluates

f (for input lengths $\ell(t_1), \ldots, \ell(t_n)$), picks a fresh register name R, queries $R := \mathtt{eval}_{\mathtt{ch}}(C, R_1, \ldots, R_n)$, and returns R.

For example: $\beta^*(sig(t_1, t_2, N))$ with $N \in \mathcal{R}$ runs $R_1 := \beta^*(t_1)$, $R_2 := \beta^*(t_2)$, computes the circuit C_{sig} that on inputs s, m of lengths $\ell(t_1), \ell(t_2)$ returns $A_{sig}(s, m, r_N)$, picks a fresh register name R, queries $R := \mathtt{eval}_{\mathtt{ch}}(C_{sig}, R_1, R_2)$, and returns R.

- Before $\tau(m)$ is invoked (when translating a bitstring m from the adversary to the protocol), for any invocation of $R := \beta^*(t)$ that has occurred so far, Sim_1 invokes $\mathtt{reveal}_{\mathtt{ch}}(R)$ if $X, red_i(S) \vdash red_i(t)$ where S is the set of terms received from the protocol so far and X is the set of all variables x^c. (This is to ensure that rules like "$\tau(e) := ek(N)$ iff e was output by $\beta(ek(N))$" work correctly.)

Using the malicious-key extractor. The simulator Sim_3 is defined like Sim_2, except for the following change: Let MKE be the malicious-key extractor for $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$; MKE exists by implementation condition 26. After invoking $t = \tau(m)$ and before sending the term t to the protocol, Sim_3 runs, for each variable x^c that has been produced by τ so far, the extractor MKE on the list of all $\mathtt{oracle}_{\mathtt{ch}}(x)$-queries and on the ciphertext c. For each message $\tilde{m}$ output by MKE, Sim_3 computes $\tau(m)$ and discards the result (but performs, if necessary, additional invocations of MKE if τ produces new variables x^c).

Furthermore, Sim_3 aborts if the following happens during some point of the execution. There is a $\mathtt{dec}_{\mathtt{ch}}(N, c)$-query that returns a message $\tilde{m}$ such that $\tilde{m}$ was *not* contained in the output of MKE in the step in which the variable x^c first occurred. (We call such an abort a *malicious-key-extraction-failure*.)

Using the fake challenger. The simulator Sim_4 is defined like Sim_3, except that it uses the fake challenger Definition 4 instead of the real challenger.

Using the ciphertext simulator directly. The simulator Sim_5 is defined like Sim_1 except for the following changes:

- For each $N \in \mathbf{N}_P$, Sim_5 maintains an instance CS_N of the ciphertext simulator (Definitions 3 and 8) for the encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ from implementation condition 25. (No real/fake challenger is not used, we access the ciphertext simulators directly.) Sim_5 also maintains an instance of the random oracle $\mathcal{O}$ and allows the ciphertext simulators to access/program $\mathcal{O}$ and to get the list of queries (see Definition 3). Furthermore, Sim_5 maintains sets $cipher_N$ and lists $plain_N$ for all $N \in \mathbf{N}_P$, all initially empty.
- Instead of invoking $A_{dec}(dk_N, c)$ in a call to $\tau(c)$, we query CS_N with $\mathtt{dec}_{\mathtt{cs}}(c)$ if $c \notin cipher_N$ and we use the value $\bot$ if $c \in cipher_N$.
- When the adversary E queries the random oracle on input x, Sim_5 replies with $\mathcal{O}(x)$.
- Instead of computing $\beta(t)$ to produce a bitstring to be sent to the adversary, the simulator first call $R := \beta'(t)$ (see below), then uses $\beta^\dagger(t)$ (see below) instead of $\beta(t)$. (The return value of $\beta'(t)$ is ignored here. However, the description of $\beta^\dagger$ refers to the outputs made by β'. The only purpose of β' is to keep track of what β^*-calls would have occurred in the execution of Sim_4.)

  The procedure β' is defined like β^* from Sim_2, except that β' does not send any oracle queries to the real challenger. (Note that β^* only performs queries that do not return answers, so these queries are not needed for computing the answer of β^*.)

  The function $\beta^\dagger$ is defined as follows:

- If there already was a call $\beta^\dagger(t)$ with the same t, the bitstring returned by that call $\beta^\dagger(t)$ is returned.
- $\beta^\dagger(ek(N))$: Query CS_N with $\mathtt{getek}_{\mathtt{cs}}()$. Return the answer.
- $\beta^\dagger(enc(ek(N), t_2, M))$ with $N, M \in \mathcal{R}$ and there was no $\mathtt{getdk}_{\mathtt{cs}}()$ query to CS_N yet: We abbreviate $t := enc(ek(N), t_2, M)$. Let R be the output that was returned when $\beta'(t)$ was invoked, and R_1 that of $\beta'(t_2)$. Query CS_N with $\mathtt{fakeenc}_{\mathtt{cs}}(R, \ell(t))$. Denote the answer with c. Append c to $cipher_N$ and append $(R \mapsto R_1)$ to $plain_N$. Return c.
- $\beta^\dagger(enc(ek(N), t_2, M))$ with $N, M \in \mathcal{R}$ and there was a $\mathtt{getdk}_{\mathtt{cs}}()$ query to CS_N: We abbreviate $t := enc(ek(N), t_2, M)$. Compute $m := \beta^\dagger(t_2)$. Let R denote the output of $\beta'(t)$, and R_1 that of $\beta'(t_2)$. Query CS_N with $\mathtt{enc}_{\mathtt{cs}}(R, m)$. Denote the answer with c. Append c to $cipher_N$ and append $(R \mapsto R_1)$ to $plain_N$. Return c.
- $\beta^\dagger(dk(N))$: Invoke $\beta^\dagger(ek(N))$ (and discard the return value). Query CS_N with $\mathtt{getdk}_{\mathtt{cs}}()$. Denote the answer with d. Do the following for each $(R \mapsto R') \in plain_N$: Find the t such that an invocation of $\beta'(t)$ returned R'. Call $m := \beta^\dagger(t)$. Send the query $\mathtt{program}_{\mathtt{cs}}(R, m)$ to CS_N. Finally, return d.
- For all other terms, $\beta^\dagger$ behaves like β.

- Before $\tau(m)$ is invoked (when translating a bitstring m from the adversary to the protocol), for any invocation of $\beta'(t)$ that has occurred so far, Sim_1 invokes $\beta^\dagger(t)$ if $X, red_i(S) \vdash red_i(t)$ where S is the set of terms received from the protocol so far and X is the set of all variables x^c.

Using fresh randomness for signatures. The simulator Sim_6 is defined like Sim_5, except that we change the definition of $\beta^\dagger$ as follows: When invoking $\beta(vk(N))$ or $\beta(sk(N))$ with $N \in \mathbf{N}_P$, a fresh key pair (vk_N, sk_N) is chosen using the key generation algorithm $\mathsf{SKeyGen}$ from implementation condition 27, unless this key pair has been chosen before, and vk_N or dk_N is returned, respectively. When invoking $\beta(sig(sk(N), t, M))$ with $N, M \in \mathbf{N}_P$, instead of computing $A_{sig}(A_{sk}(r_N), \beta(t), r_M)$ we compute $\mathsf{Sig}(A_{sk}(r_N), \beta(t_2))$ (Sig is defined in implementation condition 27).

Using a signing oracle. The simulator Sim_7 is defined like Sim_6, except that it maintains instances $\mathcal{O}_N^{\mathsf{sig}}$ of a signing oracle for the signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ from implementation condition 27 (with queries for signing, for getting the verification key, and for getting the signing key). There is an instance for any $N \in \mathbf{N}_P$, each instance is initialized when used the first time. When invoking $\beta(vk(N))$, instead of generating a key pair (vk_N, sk_N), Sim_7 queries the verification key vk_N from $\mathcal{O}_N^{\mathsf{sig}}$. When invoking $\beta(sk(N))$, instead of generating a key pair (vk_N, sk_N), Sim_7 queries the signing key sk_N from $\mathcal{O}_N^{\mathsf{sig}}$. Instead of computing $\mathsf{Sig}(\beta(sk(N)), \beta(t_2))$, Sim_7 sends a signing query with message $\beta(t_2)$ to $\mathcal{O}_N^{\mathsf{sig}}$.

C.3 The actual proof

Lemma 1 *For any (direct or recursive) invocation of $\beta^\dagger(t)$ performed by Sim_5 or Sim_7, we have that for any i, $X, red_i(S) \vdash red_i(t)$ and $S\varphi = t\varphi$ where S is the set of all terms sent by Π^c to Sim_5 (or Sim_7) up to that point and X is the set of all variables x^c. (Note: i does not have to be the number of the step in which $\beta^\dagger(t)$ is invoked.)*

Proof. We show the lemma for $X, red_i(S) \vdash red_i(t)$. $S\varphi = t\varphi$ is analogous.

In the definition of Sim_5 (or Sim_7), there are three places in which $\beta^\dagger$ is invoked: (a) When a term t is sent by the protocol Π^C, the simulator invoked $\beta^\dagger(t)$. (b) Before invoking $\tau(m)$, the simulator calls $\beta^\dagger(t)$ for terms t with $X, red_i(S) \vdash red_i(t)$. (c) $\beta^\dagger(t)$ is recursively invoked by $\beta^\dagger$.

In case (a), $t \in S$ by definition of S, hence $S \vdash t$. In case (b), we have $X, red_i(S) \vdash red_i(t)$ by assumption. Thus, to prove the lemma we only need to show that if $\beta^\dagger(t)$ is recursively invoked by $\beta^\dagger(t')$, and if $X, red_i(S) \vdash red_i(t'')$ for all earlier invocations $\beta^\dagger(t'')$ (including the case $t'' = t'$), then we have $X, red_i(S) \vdash red_i(t)$

We distinguish the different cases for t':

Case 1: *"$t' = ek(N)$".*
In this case, $\beta^\dagger(t')$ does not perform any recursive calls.

Case 2: *"$t' = enc(ek(N), t_2, M)$ with $N, M \in \mathcal{R}$ and there was no $\texttt{getdk}_{\texttt{cs}}()$ query to CS_N yet".*
In this case, $\beta^\dagger(t')$ invokes $\beta^\dagger(ek(N))$. Hence $t = ek(N)$. Since $red_i(t') = enc(ek(N), \dots) \vdash ek(N) = red_i(t)$, we have $X, red_i(S) \vdash red_i(t)$.

Case 3: *"$t' = enc(ek(N), t_2, M)$ with $N, M \in \mathcal{R}$ and there was a $\texttt{getdk}_{\texttt{cs}}()$ query to CS_N".*
A $\texttt{getdk}_{\texttt{cs}}()$-query to CS_N query is only performed by $\beta^\dagger(dk(N))$. Thus $X, red_i(S) \vdash red_i(dk(N)) = dk(N)$. With $X, red_i(S) \vdash red_i(t') = enc(ek(N), red_i(t_2), M)$, this implies $X, red_i(S) \vdash red_i(t_2)$ and $X, red_i(S) \vdash ek(N) = red_i(ek(N))$. $\beta^\dagger(t_2)$ and $\beta^\dagger(ek(N))$ are the only recursive calls.

Case 4: *"$t' = dk(N)$".*
Then $X, red_i(S) \vdash red_i(t') = dk(N)$. An invocation of $\beta^\dagger(dk(N))$ invokes $\beta^\dagger(ek(N))$ and additionally invokes $\beta^\dagger(t)$ only if $(R \mapsto R') \in plain_N$ where R' was returned by $\beta'(t)$. The case $t = ek(N)$ is handled as in Case 2. Thus we can assume that $(R \mapsto R') \in plain_N$ where R' was returned by $\beta'(t)$. Such an $(R \mapsto R')$ is only appended to $plain_N$ by an invocation $\beta^\dagger(enc(ek(N), t_2, M))$ with $R = \beta'(t_2)$. Thus $X, red_i(S) \vdash red_i(enc(ek(N), t_2, M)) = enc(ek(N), red_i(t_2), M)$ and $t = t_2$. Together with $X, red_i(S) \vdash red_i(t') = dk(N)$ we get $X, red_i(S) \vdash red_i(t_2) = red_i(t)$.

Case 5: *"$t' = sig(sk(N), t_2, M)$ or $sk(N)$ with $M \in \mathbf{N}_P$".*
In this case, $\beta^\dagger(t')$ invokes only $\beta^\dagger(vk(N))$. This handled analogously to Case 2.

Case 6: *"$t' \in \{pair(t_1, t_2), string_0(t_1), string_1(t_2)\}$ with $M \in \mathbf{N}_P$".*
Since $X, red_i(S) \vdash red_i(t')$, we have $X, red_i(S) \vdash red_i(t)$ or $X, red_i(S) \vdash red_i(t_1), red_i(t_2)$. And $\beta^\dagger(t')$ invokes only $\beta^\dagger(t)$ or $\beta^\dagger(t_1), \beta^\dagger(t_2)$, respectively (see the definition of β, since $\beta^\dagger$ by definition behaves like β in these cases).

Case 7: *"All other cases.".*
In these cases, $\beta^\dagger$ does not perform recursive invocations to $\beta^\dagger$ (see the definition of β, since $\beta^\dagger$ by definition behaves like β in these cases). □

Lemma 2 *The full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim_7}$ are computationally indistinguishable.*

Proof. We prove the lemma by showing the indistinguishability of the full traces of any two consecutive simulators. First, we have:

Claim 1 *$H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathrm{p},Sim}$ have the same distribution.*

To show that claim, note that the only difference between Sim and Sim_1 is that the randomness for the key generation, the encryption, and the signing is chosen by the algorithms KeyGen, SKeyGen, Enc, and Sig in Sim_1, while Sim uses nonces r_N instead. However, from protocol conditions 1, 2, 3, 4, it follows that Sim never uses a given randomness r_N twice (note that, since $N \in \mathcal{R}$, τ does not access r_N either). Hence the full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_1}$ are indistinguishable. This shows Claim 1.

In order to state the next indistinguishability, we first need to define two events:

By DecryptOwn we denote the event that an $\mathtt{dec}_{\mathtt{ch}}(N,c)$-query to the real/fake challenger returns `forbidden` (due to the fact that $c \in cipher_N$). (Note: the event DecryptOwn is only well defined for the simulators Sim_2 and Sim_4 who use the real/fake challenger.)

By BetaOutputLost, we denote the event that there exist R, t such that $\tau(reg_R)$ is invoked, reg_R was not yet revealed at that point (using a $\mathtt{reveal}_{\mathtt{ch}}(R)$-query), R was returned by $\beta^*(t)$, and $t \in \{ek(N), dk(N), vk(N), sk(N), enc(\ldots, N), sig(\ldots, N) : N \in \mathbf{N}_P\}$. (Note: the event BetaOutputLost is only meaningful for the simulator Sim_2. In a hybrid execution of Sim_4, the fake challenger is used, so non-revealed registers will not contain bitstrings.)

Claim 2 *If* BetaOutputLost *and* DecryptOwn *have negligible probability in the hybrid execution of Sim_2, then $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_1}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_2}$ are statistically indistinguishable.*

To prove this claim, first notice the effect of $R := \beta^*(t)$ is to make the real challenger compute $\beta(t)$ and to put the result of this computation into the register reg_R. A call $\beta(t)$ is replaced by $R := \beta^*(t)$, $\mathtt{reveal}_{\mathtt{ch}}(R)$; the result of this query is the same as $\beta(t)$ would have returned.

However, in Sim_1, τ refers in several places to the outputs of the calls to β. E.g., "$\tau(dk(N))$ if k has earlier been output by $\beta(ek(N))$". This rule now (in Sim_2) reads "$\tau(dk(N))$ if k has earlier been output by $\beta^*(ek(N))$" However, there $\beta^*(t)$ is not computed for every term for which $\beta(t)$ would have been computed in Sim_1. Some of the $\beta(t)$-results of Sim_1 are now just contained in the registers of the real challenger but not revealed to Sim_2. I.e., $\tau(m)$ will behave differently if m is contained in the register reg_R where R was returned by $\beta^*(t)$ for some $t \in \{ek(N), dk(N), vk(N), sk(N), enc(\ldots, N), sig(\ldots, N) : N \in \mathbf{N}_P\}$, and that register was not yet revealed (using a $\mathtt{reveal}_{\mathtt{ch}}(R)$-query). I.e., $\tau(m)$ only behaves differently if the event BetaOutputLost occurs.

Finally, we have changed the way messages are decrypted by τ: Sim_2 uses a $\mathtt{dec}_{\mathtt{ch}}(N,c)$-query for that. This query fails if $c \in cipher_N$. But unless the event DecryptOwn occurs, $\mathtt{dec}_{\mathtt{ch}}(N,c)$-queries decrypt correctly.

Thus the statistical distance between $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_1}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_2}$ is bounded by the probability that BetaOutputLost or DecryptOwn occurs in the execution of Sim_2. This shows Claim 2.

Claim 3 *The full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_2}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_3}$ are statistically indistinguishable.*

The executions of Sim_2 and Sim_3 differ only in case of a malicious-key-extraction-failure. But malicious-key-extraction-failures happen only with negligible probability due to the malicious-key extractability of (KeyGen, Enc, Dec) (implementation condition 26).

Claim 4 *The full traces $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_3}$ and $H\text{-}Trace_{\mathbf{M},\Pi_\mathsf{p},Sim_4}$ are computationally indistinguishable.*

This claim follows directly from the PROG-KDM security of (KeyGen, Enc, Dec) from implementation condition 25.

Claim 5 *The full traces H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_4}$ and H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_5}$ have the same distribution.*

To show this claim, we observe the following: In Sim_5, the function β' behaves like the function β^* from Sim_4, except that it does not query the fake challenger. Furthermore, in Sim_4, we have queries $\texttt{reveal}_{\texttt{ch}}(R)$ with $R := \beta'(t)$, these are replaced by the computation $\beta^\dagger(t)$. By induction over the recursive definition of $\beta^\dagger$ and by comparing it to the definition of `FCRetrieve`, we see that $\beta^\dagger(t)$ always returns what $\texttt{FCRetrieve}(R)$ would return for $R := \beta'(t)$. Since $\texttt{reveal}_{\texttt{ch}}(R)$ returns $\texttt{FCRetrieve}(R)$, it follows that $\beta^\dagger(t)$ gives the same result as $\texttt{reveal}_{\texttt{ch}}(R)$ with $R := \beta'(t)$. This shows Claim 5.

Claim 6 *The full traces H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_5}$ and H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_6}$ have the same distribution.*

This claim is shown analogously to Claim 1.

Claim 7 *The full traces H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_6}$ and H-Trace$_{\mathbf{M},\Pi_\mathsf{p},Sim_7}$ have the same distribution.*

This claim follows from the fact that in Sim_7 all verification/signing key generation and all signing operations are outsourced to a signing oracle which performs them in the same way as they were performed by Sim_6.

Claim 8 *In a hybrid execution with Sim_2, the probability of* BetaOutputLost *is negligible.*

To prove this claim, we first consider a modification of the simulators $Sim_2, \ldots, Sim_7$. We define the simulator Sim_2^* to behave like Sim_2, except that we add a post-processing phase after the execution: In this phase, for any $R := \beta^*(t)$ that was computed during the execution (including recursive calls to β^*), the simulator queries $\texttt{reveal}_{\texttt{ch}}(R)$ from the real challenger.

Analogously we define Sim_4^*.

Sim_5^* has a different post-processing phase: For any invocation $\beta'(t)$ during the execution, Sim_5 queries $\beta^\dagger(t)$ in the post-processing phase.

Sim_6^* and Sim_7^* are defined analogously.

Since BetaOutputLost is only meaningful for Sim_2 (and of course Sim_2^*), we need to define analogous events for the other simulators.

For Sim_2^* and Sim_4^*, we define the following variant: Let $\mathsf{BetaOutputLost}_A$ denote the event that there exist a bitstring m, a register R, and a term t such that the $\texttt{reveal}_{\texttt{ch}}(R)$ query in the post-processing phase returns m, and there is an invocation $\tau(m)$ that occurred before any $\texttt{reveal}_{\texttt{ch}}(R)$ query, and an invocation $\beta^*(t)$ returned R, and $t \in \{ek(N), dk(N), vk(N), sk(N), enc(\ldots, N), sig(\ldots, N) : N \in \mathbf{N}_P\}$.

For Sim_5^*, Sim_6^*, and Sim_7^*, we define the following variant: Let $\mathsf{BetaOutputLost}_B$ denote the event that there exist a bitstring m and a term t such that the invocation $\beta^\dagger(t)$ in the post-processing phase returns m, and there is an invocation $\tau(m)$ that occurred before any invocation $\beta^\dagger(t)$, and $t \in \{ek(N), dk(N), vk(N), sk(N), enc(\ldots, N), sig(\ldots, N) : N \in \mathbf{N}_P\}$.

And finally, for Sim_5^*, we define the following variant: Let $\mathsf{BetaOutputLost}_B$ denote the event that there exist a bitstring m and a term t such that the top-level invocation $\beta^\dagger(t)$ in the post-processing phase returns m, and there is an invocation $\tau(m)$ that occurred before any top-level invocation $\beta^\dagger(t)$, and $t \in \{ek(N), dk(N), vk(N), sk(N), enc(\ldots, N), sig(\ldots, N) : N \in \mathbf{N}_P\}$. By a "top-level invocation" of $\beta^\dagger$ we mean an invocation that was not recursively invoked by $\beta^\dagger$. (I.e., the top-level invocations of $\beta^\dagger$ correspond directly to the $\texttt{reveal}_{\texttt{ch}}$-queries in Sim_4^*.) The only difference between $\mathsf{BetaOutputLost}_B$ and $\mathsf{BetaOutputLost}_C$ is that in the latter we only consider top-level invocations.

Let $\Pr[X : Sim_i]$ denote the probability that the event X occurs in the hybrid execution with Sim_i. Then $\Pr[\mathsf{BetaOutputLost} : Sim_2] = \Pr[\mathsf{BetaOutputLost} : Sim_2^*] =$

$\Pr[\mathsf{BetaOutputLost}_A : Sim_2^*]$. Furthermore $\Pr[\mathsf{BetaOutputLost}_A : Sim_2^*] \approx \Pr[\mathsf{BetaOutputLost}_A : Sim_4^*] = \Pr[\mathsf{BetaOutputLost}_C : Sim_5^*]$ and $\Pr[\mathsf{BetaOutputLost}_B : Sim_5^*] = \Pr[\mathsf{BetaOutputLost}_B : Sim_6^*] = \Pr[\mathsf{BetaOutputLost}_B : Sim_7^*]$ where $\approx$ denotes a negligible difference.

These equalities follow analogously to Claims 4–7. Thus $\Pr[\mathsf{BetaOutputLost} : Sim_2] \approx \Pr[\mathsf{BetaOutputLost}_B : Sim_7^*]$.

Furthermore, we claim that $\Pr[\mathsf{BetaOutputLost}_C : Sim_5^*] = \Pr[\mathsf{BetaOutputLost}_B : Sim_5^*]$. To show this, it is suffient to show that if $\beta^\dagger(t)$ is invoked (possibly recursively), then there is also a top-level invocation of $\beta^\dagger(t)$ in the same step. For this, fix a term t such that $\beta^\dagger(t)$ was invoked in the i-th step. By Lemma 1, we have that $X, red_i(S) \vdash red_i(t)$. If $\beta^\dagger(t)$ was invoked recursively, there must have been a $\beta'(t)$-call in the same invocation. Furthermore, by construction of Sim_5^* (see the description of Sim_5), we have that for any $\beta'(t)$-call with $X, red_i(S) \vdash red_i(t)$, $\beta^\dagger(t)$ is invoked before the invocation of $\tau(m)$ (i.e., still in the same step). Thus a top-level invocation of $\beta^\dagger(t)$ occurs. Thus $\Pr[\mathsf{BetaOutputLost}_C : Sim_5^*] = \Pr[\mathsf{BetaOutputLost}_B : Sim_5^*]$.

Thus $\Pr[\mathsf{BetaOutputLost} : Sim_2] \approx \Pr[\mathsf{BetaOutputLost} : Sim_7^*]$.

Assume that $\Pr[\mathsf{BetaOutputLost}_B : Sim_7^*]$ is not negligible. Then one of the following occurs with not-negligible probability:

- $\mathsf{BetaOutputLost}_B$ occurs with $t = ek(N)$: By construction, $\beta^\dagger(t) = \beta^\dagger(ek(N))$ returns the result of querying CS_N with $\mathtt{getek}_{\mathtt{cs}}()$. I.e., $m = ek_N$ where ek_N is the encryption key maintained by the ciphertext simulator CS_N. Before the invocation of $\tau(m)$, there was no call to $\beta^\dagger(t) = \beta^\dagger(ek(N))$. Since $\beta^\dagger(dk(N))$ and $\beta(enc(\ldots, N))$ recursively invoke $\beta^\dagger(ek(N))$, it follows that there was no call to $\beta^\dagger(ek(N))$ either. Since $\mathtt{getek}_{\mathtt{cs}}$-, $\mathtt{getdk}_{\mathtt{cs}}$-, and $\mathtt{fakeenc}_{\mathtt{cs}}$-, and $\mathtt{enc}_{\mathtt{cs}}$-queries to CS_N are only performed by $\beta^\dagger(ek(N))$, $\beta^\dagger(dk(N))$, $\beta^\dagger(enc(\ldots, N))$, it follows that before the invocation of $\tau(m)$ none of these queries was sent to CS_N. So before $\tau(m)$, only $\mathtt{dec}_{\mathtt{cs}}$-queries may have been sent to CS_N. It is easy to see that in a PROG-KDM secure encryption scheme, the encryption key cannot be guessed with not-negligible probability without making any encryption or decryption queries (that would imply that two independently chosen keys are equal with not-negligible probability, allowing to break the scheme with not-negligible probability). Due to implementation condition 8 (which requires that a ciphertext is tagged with its encryption key), decryption queries also do not help in guessing an encryption key. Thus the probability that a $\tau(m)$ query with $m = ek_N$ is performed before any $\mathtt{getek}_{\mathtt{cs}}$-, $\mathtt{getdk}_{\mathtt{cs}}$-, and $\mathtt{fakeenc}_{\mathtt{cs}}$-, and $\mathtt{enc}_{\mathtt{cs}}$-queries to CS_N is negligible. Thus $\mathsf{BetaOutputLost}_B$ with $t = ek(N)$ cannot occur with not-negligible probability.

- $\mathsf{BetaOutputLost}_B$ occurs with $t = dk(N)$: By construction, $\beta^\dagger(t) = \beta^\dagger(dk(N))$ returns the result of querying CS_N with $\mathtt{getdk}_{\mathtt{cs}}()$. I.e., $m = dk_N$ where dk_N is the decryption key maintained by the ciphertext simulator CS_N. Before the invocation of $\tau(m)$, there was no call to $\beta^\dagger(t) = \beta^\dagger(dk(N))$. Since $\mathtt{getdk}_{\mathtt{cs}}$-queries to CS_N are only performed by $\beta^\dagger(dk(N))$, it follows that before the invocation of $\tau(m)$ no $\mathtt{getdk}_{\mathtt{cs}}$-query was sent to CS_N. Being able to guess $m = dk_N$ (with not-negligible probability) without performing $\mathtt{getdk}_{\mathtt{cs}}$-queries would contradict PROG-KDM security. Thus $\mathsf{BetaOutputLost}_B$ with $t = dk(N)$ cannot occur with not-negligible probability.

- $\mathsf{BetaOutputLost}_B$ occurs with $t = vk(N)$: Analogous to $t = ek(N)$. (Except that we consider the signing oracle instead of the ciphertext simulator, and signing instead of encrypting, and do not have to deal with the decryption-case.)

- $\mathsf{BetaOutputLost}_B$ occurs with $t = sk(N)$: Analogous to $t = dk(N)$.

- $\mathsf{BetaOutputLost}_B$ occurs with $t = enc(ek(M), t', N)$: By construction, $m = \beta^\dagger(t) = \beta^\dagger(enc(ek(M), t', N))$ returns either the result of querying CS_N via an $\mathtt{enc}_{\mathtt{cs}}$- or $\mathtt{fakeenc}_{\mathtt{cs}}$-query, or the result of invoking $\mathsf{Enc}(A_{ek}(r_M), \beta^\dagger(t))$. (Depending on whether $M \in \mathbf{N}_P$ or $M \in \mathbf{N}_E$.) Since $\tau(m)$ is invoked before the first $\beta^\dagger(t)$-invocation, $\tau(m)$ is invoked before the $\mathtt{enc}_{\mathtt{cs}}$- or $\mathtt{fakeenc}_{\mathtt{cs}}$-query or computation of $\mathsf{Enc}(A_{ek}(r_M), \beta^\dagger(t))$ is performed. This violated the unpredictability of $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ (implementation condition implementation condition 28).[14] Thus $\mathsf{BetaOutputLost}_B$ with $t = enc(\ldots, N)$ cannot occur with not-negligible probability.
- $\mathsf{BetaOutputLost}_B$ occurs with $t = sig(\ldots, N)$. Analogous to $t = enc(\ldots, N)$, except that we use the unpredictability of the signature scheme (implementation condition 29).

Thus $\Pr[\mathsf{BetaOutputLost}_B : Sim_7^*]$ is negligible, and – since $\Pr[\mathsf{BetaOutputLost} : Sim_2] \approx \Pr[\mathsf{BetaOutputLost} : Sim_7^*]$ – Claim 8 is shown.

Claim 9 *In a hybrid execution with Sim_2, the probability of* DecryptOwn *is negligible.*

To show this claim, we first show that DecryptOwn implies BetaOutputLost. Assume that DecryptOwn occurs. Then a $\mathtt{dec}_{\mathtt{ch}}(N, c)$-query to the real challenger has been performed with $c \in cipher_N$. By construction of Sim_2, such a query is only performed by $\tau(c)$, and only if c is of type ciphertext and c was not output earlier by any $\mathtt{reveal}_{\mathtt{ch}}(R)$-query (for any R). Furthermore, by definition of the real challenger, $c \in cipher_N$ implies that there was an $R := \mathtt{enc}_{\mathtt{ch}}(N, p)$-query for some R, p and that $reg_R = c$. An $R := \mathtt{enc}_{\mathtt{ch}}(N, p)$-query is only performed by an invocation $\beta^*(enc(ek(N), t', M))$ for some $t \in \mathbf{T}_x, M \in \mathbf{N}_P$.

Thus, there exist $m := c$ and R such that $\tau(m)$ is invoked, reg_R was not yet revealed at that point, R was returned by $\beta^*(enc(ek(N), t', M))$ for some $t \in \mathbf{T}_x, M \in \mathbf{N}_P$. By definition of BetaOutputLost, this means BetaOutputLost occurred. So DecryptOwn implies BetaOutputLost. By Claim 8, BetaOutputLost occurs with negligible probability, this DecryptOwn occurs with negligible probability. This shows Claim 9.

We can now finish the proof of Lemma 2. By Claims 1–7, we have that $H\text{-}Trace_{\mathbf{M},\Pi_{\mathtt{p}},Sim}$ and $H\text{-}Trace_{\mathbf{M},\Pi_{\mathtt{p}},Sim_7}$ are computationally indistinguishable if BetaOutputLost and DecryptOwn have negligible probability in the hybrid execution of Sim_2. The latter is the case by Claims 8 and 9. Thus Lemma 2 follows. □

Lemma 3 *In the hybrid execution of Sim, we have for any i, j, and any term t, we have $red_i(t)\varphi = red_j(t)\varphi$ where φ is the final substitution.*

Proof. Let n be the last step of the execution, and $\sigma(x^c) := garbageEnc(N^{A_{ekof}(c)}, N^c)$ (as in the definition of φ). It is sufficient to show $red_i(t)\varphi = red_n(t)\varphi$ for $i < n$ since then $red_i(t)\varphi = red_n(t)\varphi = red_j(t)\varphi$.

We have that red_n and red_i behave differently only on inputs x^c such that $dk(N^{A_{ekof}(c)})$ did not occur up to step i of the execution but occurred up to step n.

Then $red_i(x^c)\varphi = \varphi(x^c) \overset{(*)}{=} red_n(x^c)\sigma$. Since $red_n(x^c)$ only contains variables for which $red_n(x^c) = x^c$ and thus $\varphi(x^c) = \sigma(x^c)$, we have that $red_n(x^c)\varphi = red_n(x^c)\sigma$. Hence $red_i(x^c)\varphi = red_n(x^c)\varphi$. □

[14] In the case of an $\mathtt{enc}_{\mathtt{cs}}$- or $\mathtt{fakeenc}_{\mathtt{cs}}$-query, this is not fully immediate because the ciphertext simulator is not required to compute the answer to these queries using Enc. However, if it was possible to predict the value of a ciphertext returned by the ciphertext simulator, this would imply that it is also possible to predict the value of a ciphertext returned by the real challenger (due to PROG-KDM security). The latter actually uses Enc, so predicting its ciphertexts would violate the unpredictability of $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

Lemma 4 *Sim is consistent for* $\mathbf{M}$, Π, A, *and for every polynomial* p.

Proof. By definition, to prove the consistency of *Sim*, we need to show that whenever *Sim* is asked a question Q, then the answer to Q is yes iff $\mathrm{eval}(Q\varphi) \neq \bot$ where φ is the final substitution output by *Sim* in the end. By definition of *Sim*, *Sim* answers yes iff $red_i(Q) \neq \bot$ where i refers to the current step of the execution (the number of the node in question along the path takes in the protocol tree). Thus we have to show that $red_i(Q) \neq \bot$ iff $\mathrm{eval}(Q\varphi) \neq \bot$. To show that, we first need the following auxiliary claim:

Claim 1 *For any subterm* $f(Q_1, \ldots, Q_n)$ *of* Q *with* $f/n \in \mathbf{C} \cup \mathbf{D}$, *we have that* $\mathrm{eval}_f(red_i(Q_1), \ldots, red_i(Q_n))\varphi \equiv \mathrm{eval}_f(red_i(Q_1)\varphi, \ldots, red_i(Q_n)\varphi)$. *Here* $\equiv$ *denotes that the lhs is defined iff the rhs is and that both are equal in that case.*

To show this claim, first remember that $\varphi(x^c)$ is always of the form $enc(ek(N^{A_{ekof}(c)}), \cdot, N^c)$. Furthermore, if x^c occurs in Q, then $\varphi(x^c)$ does not occur in Q. (This stems from the fact that if $\tau(c)$ translates c to x^c, then it always does so. Hence no term of the form $enc(\ldots, N^c)$ will be produced by τ.) And $red_i(Q_n) \in \mathbf{T}_x$ if defined.

These observations are sufficient to check that the claim holds for each $f \neq dec$. (For the case $f = isenc$, remember that we introduced (on page 27) the rules $isenc(x^c) = x^c$ and $ekof(x^c) = ek(N^{A_{ekof}(c)})$ in addition to the destructor rules in given in Figure 1.

Now consider the case $f = dec$. We abbreviate $\tilde{Q}_j := red_i(Q_j)$ for $j = 1, 2$. We need to show

$$\mathrm{eval}_{dec}(\tilde{Q}_1, \tilde{Q}_2)\varphi \equiv \mathrm{eval}_{dec}(\tilde{Q}_1\varphi, \tilde{Q}_2\varphi). \tag{1}$$

If $Q_1 \neq dk(\cdot)$, then also $\tilde{Q}_1\varphi \neq dk(\cdot)$, and (1) follows. Thus we can assume $Q_1 = dk(N)$ for some $N \in \mathbf{N}$. If $\tilde{Q}_2$ is neither of the form $\tilde{Q}_2 = enc(ek(N), \ldots)$ or $\tilde{Q}_2 = x^c$, then $\mathrm{eval}_{dec}(\tilde{Q}_1, \tilde{Q}_2) = \bot$ and $\mathrm{eval}_{dec}(\tilde{Q}_1\varphi, \tilde{Q}_2\varphi) = \bot$ and (1) follows. Thus we can assume that $\tilde{Q}_2 = enc(ek(N), t, M)$ with $N, M \in \mathbf{N}$ or $\tilde{Q}_2 = x^c$. In the first case, we have $\mathrm{eval}_{dec}(\tilde{Q}_1, \tilde{Q}_2)\varphi \equiv t\varphi \equiv \mathrm{eval}_{dec}(dk(N), enc(ek(N), t\varphi, M)) \equiv \mathrm{eval}_{dec}(\tilde{Q}_1\varphi, \tilde{Q}_2\varphi)$, so (1) holds in this case.

Thus we can assume $\tilde{Q}_2 = x^c$ for some c. By definition of red_i, we have $red_i(x^c) = enc(\ldots)$ or $red_i(x^c) = garbageEnc(\ldots)$ if the term $dk(N^e)$ with $e := A_{ekof}(c)$ occurred at some node prior to the current one. Thus $\tilde{Q}_2 = x^c$ implies that $dk(N^e)$ did not occur at any prior node. Since Q_1 is the term at one of the predecessors of the current node, $Q_1 \neq dk(N^e)$ and hence $\tilde{Q}_1\varphi \neq dk(N^e)$. Furthermore, by construction of φ, we have $\varphi(x^c) = enc(ek(N^e), \ldots)$ or $\varphi(x^c) = enc(ek(N^e), N^c)$. Thus $dec(\tilde{Q}_1\varphi, \tilde{Q}_2\varphi) = \bot$. Hence $\mathrm{eval}_{dec}(\tilde{Q}_1, \tilde{Q}_2)\varphi \equiv \mathrm{eval}_{dec}(\tilde{Q}_1, x^e)\varphi \equiv \bot \equiv \mathrm{eval}_{dec}(\tilde{Q}_1\varphi, \tilde{Q}_2\varphi)$. Thus (1) holds also in this last case. This shows Claim 1.

We can now prove the following claim which is already almost the result we need:

Claim 2 *For any subterm* Q' *of* Q, *we have that* $red_i(Q')\varphi \equiv \mathrm{eval}(Q'\varphi)$.

We show this claim by structural induction over Q'. For $Q' = N \in \mathbf{N}$, the claim follows from $red_i(Q')\varphi \equiv N \equiv \mathrm{eval}(Q'\varphi)$.

For $Q' = f(Q_1, \ldots, Q_n)$ with $f \in \mathbf{C} \cup \mathbf{D}$, we have

$$\begin{aligned} red_i(Q')\varphi &\equiv \mathrm{eval}_f(red_i(Q_1), \ldots, red_i(Q_n))\varphi \\ &\overset{(*)}{\equiv} \mathrm{eval}_f(red_i(Q_1)\varphi, \ldots, red_i(Q_n)\varphi) \\ &\overset{\text{IH}}{\equiv} \mathrm{eval}_f(\mathrm{eval}(Q_1\varphi), \ldots, \mathrm{eval}(Q_n\varphi)) \\ &\equiv \mathrm{eval}(Q'\varphi) \end{aligned}$$

where $(*)$ uses Claim 1 and IH stands for the induction hypothesis.

Finally, if $Q' = x^c$, we have to show $red_i(x^c)\varphi \equiv \mathrm{eval}(\varphi(x^c))$. Since $\varphi(x^c) \in \mathbf{T}_x$, we have $\mathrm{eval}(\varphi(x^c)) \equiv \varphi(x^c)$. Thus it suffices to show $red_i(x^c)\varphi = \varphi(x^c)$ (notice that we use $=$ here, because neither red_i nor φ fail). Let $e := A_{ekof}(c)$ and $d := A^{-1}_{ekofdk}(e)$. (e is defined by implementation condition 8. d is defined if $dk(N^e)$ did not occur at any prior node to the current one because $dk(N^e)$ must then have been produced by $\tau(d)$ with $e = A_{ekofdk}(d)$.)

We distinguish three cases when showing $red_i(x^c)\varphi = \varphi(x^c)$:

Case 1: *"$dk(N^e)$ did not occur at any prior node to the current one".*
Then by definition of red_i, we have $red_i(x^c) = x^c$ and thus $red_i(x^c)\varphi = \varphi(x^c)$.

Case 2: *"$dk(N^e)$ did occur at a prior node, and $A_{dec}(d, c) = \bot$".*
Then by definition of red_i, we have $red_i(x^c) = garbageEnc(ek(N^e), N^c)$. Hence $red_i(x^c)\varphi = garbageEnc(ek(N^e), N^c)$. Furthermore, $\varphi(x^c) = red_n(x^c)\sigma$ where n is the last step of the execution, and σ a substitution (σ and n were defined in the definition of the final substitution φ.) We have $red_n(x^c) = garbageEnc(ek(N^e), N^c)$. Thus $\varphi(x^c) = garbageEnc(ek(N^e), N^c) = red_i(x^c)\varphi$.

Case 3: *"$dk(N^e)$ did occur at a prior node, and $A_{dec}(d, c) \neq \bot$".*
In this case, $red_i(x^c)\varphi = enc(ek(N^e), red_i(\tau(A_{dec}(d, c))), N^c)\varphi = enc(ek(N^e), red_i(\tau(A_{dec}(d, c)))\varphi, N^c)$. And $\varphi(x^c) = red_n(x^c)\sigma$ where n is the last step of the execution, and σ a substitution (σ and n were defined in the definition of the final substitution φ) and $red_n(x^c) = enc(ek(N^e), red_n(\tau(A_{dec}(d, c)))\sigma, N^c)$. Thus, to show that $red_i(x^c)\varphi = \varphi(x^c)$, it suffices to show that $red_i(\tau(A_{dec}(d, c)))\varphi = red_n(\tau(A_{dec}(d, c)))\sigma$.

We have $red_i(\tau(A_{dec}(d, c)))\varphi = red_n(\tau(A_{dec}(d, c)))\varphi$ by Lemma 3. And by definition of red_n, $red_n(\tau(A_{dec}(d, c)))$ only contains variables $x^{c'}$ with $red_n(x^{c'}) = c'$ and hence with $\varphi(x^{c'}) = red_n(x^{c'}) = \sigma^{c'}$. Thus we have $red_n(\tau(A_{dec}(d, c)))\varphi = red_n(\tau(A_{dec}(d, c)))\sigma$ and $red_i(\tau(A_{dec}(d, c)))\varphi = red_n(\tau(A_{dec}(d, c)))\sigma$ follows.

Thus, if $Q' = x^c$, we have $red_i(Q')\varphi \equiv \mathrm{eval}(Q'\varphi)$.

This shows Claim 2.

The lemma now follows: Since $red_i(Q) \neq \bot$ iff $red_i(Q)\varphi \neq \bot$, we have $red_i(Q) \neq \bot$ iff $\mathrm{eval}(Q\varphi) \neq \bot$ by Claim 2. As explained before Claim 1, this implies that Q is answered with yes iff $\mathrm{eval}(Q\varphi) \neq \bot$, thus *Sim* is consistent. □

Lemma 5 *Sim is indistinguishable for* $\mathbf{M}$, Π, A, *and for every polynomial* p.

Proof. We will first show that when fixing the randomness of the adversary and the protocol, the node trace $Nodes^p_{\mathbf{M},A,\Pi_\mathrm{p},E}$ in the computational execution and the node trace $H\text{-}Nodes_{\mathbf{M},\Pi_\mathrm{p},Sim}$ in the hybrid execution are equal. Hence, fix the variables r_N for all $N \in \mathbf{N}_P$, fix a random tape for the adversary, and for each non-deterministic node ν fix a choice e_ν of an outgoing edge.

We assume that the randomness is chosen such that all bitstrings r_N, $A_{ek}(r_N)$, $A_{dk}(r_N)$, $A_{vk}(r_N)$, $A_{sk}(r_N)$, $A_{enc}(e, m, r_N)$, and $A_{sig}(s, m, r_N)$ computed during the execution are all pairwise distinct. It is easy to see from the implementation conditions that this holds with overwhelming probability.

In the following, we designate the values f_i and ν_i in the computational execution by f'_i and ν'_i, and in the hybrid execution by f^C_i and ν^C_i. Let s'_i denote the state of the adversary E in the computational model, and s^C_i the state of the simulated adversary in the hybrid model.

Claim 1: In the hybrid execution, for any $b \in \{0, 1\}^*$, $\beta(\tau(b)) = b$.

This claim follows by induction over the recursive evaluation of τ.

Claim 2: In the hybrid execution, for any term t stored at node ν_i, $\beta(red_i(t)) \neq \bot$.

By induction on the structure of t and by checking that all terms that would lead to $\beta(\dots) = \bot$ are never produced by τ or the protocol. And using the fact that $red_i(t) \neq t$ since otherwise the simulator would have answered no for the question $Q := t$ so that t would not have been stored at the node in the first place.

Claim 3: For all terms $t \notin \mathcal{R}$ that occur in the hybrid execution, $\tau(\beta(t)) = t$.

By induction on the structure of t and using the assumption that r_N, $A_{ek}(r_N)$, $A_{dk}(r_N)$, $A_{vk}(r_N)$, $A_{sk}(r_N)$, as well as all occuring encryptions and signatures are pairwise distinct for all $N \in \mathcal{N}$. For terms t that contain randomness nonces, note that by protocol conditions 1–4, randomness nonces never occur outside the last argument of E-, sig-, ek-, dk-, vk-, or sk-terms.

Claim 4: In the hybrid execution, at any computation node $\nu = \nu_i$ with constructor or destructor F and arguments $\bar{\nu}_1, \dots, \bar{\nu}_n$ the following holds: Let t_j^* be the term stored at node $\bar{\nu}_j$ (i.e., $t_j^* = f_i'(\bar{\nu}_j)$). Then $\beta(\text{eval}_F(t_1, \dots, t_n)) = A_F(\beta(t_1), \dots, \beta(t_n))$ with $t_j := red_i(t_j^*)$. Here the left hand side is defined iff the right hand side is.

We show Claim 4. We distinguish the following cases:

Case 1: *"$F = ek$".*
Note that by protocol condition 1, we have $t_1 \in \mathbf{N}_P$. Then $\beta(ek(t_1)) = A_{ek}(r_{t_1}) = A_{ek}(\beta(t_1))$.

Case 2: *"$F \in \{dk, vk, sk\}$".*
Analogous to the case $F = ek$.

Case 3: *"$F \in \{pair, fst, snd, string_0, string_1, unstring_0, unstring_1, empty\}$".*
Claim 4 follows directly from the definition of β.

Case 4: *"$F = isek$".*
If $t_1 = ek(t_1')$, we have that $t_1' = N \in \mathcal{R}$ or $t_1' = N^m$ where m is of type ciphertext (as other subterms of the form $ek(\cdot)$ are neither produced by the protocol nor by τ). In both cases, $\beta(ek(t_1'))$ is of type encryption key. Hence $\beta(isek(t_1)) = \beta(ek(t_1')) = A_{isek}(\beta(ek(t_1'))) = A_{isek}(\beta(t_1))$. If t_1 is not of the form $ek(\cdot)$, then $\beta(t_1)$ is not of type public key (this uses that τ only uses N^m with m of type public key inside a term $ek(N^m)$). Hence $\beta(isek(t_1)) = \bot = A_{isek}(\beta(t_1))$.

Case 5: *"$F \in \{isvk, isdk, issk, issig\}$".*
Similar to the case $F = isek$.

Case 6: *"$F = ekofdk$".*
If $t_1 = dk(N)$ for $N \in \mathcal{R}$, then $A_{ekofdk}(\beta(dk(N)) = A_{ekofdk}(A_{dk}(r_N)) \overset{(*)}{=} A_{ek}(r_N) = \beta(ek(N)) = \beta(ekofdk(t_1))$. Here $(*)$ uses implementation condition 15. If $t_1 = dk(N^e)$, then $A_{ekofdk}(\beta(sk(N^e))) = A_{ekofdk}(A^{-1}_{ekofdk}(e)) = e = \beta(ek(N^e)) = \beta(ekofdk(t_1))$. If t_1 is not of the form $dk(\cdot)$, then $\beta(t_1)$ is not of type decryption key, and thus $A_{ekofdk}(\beta(t_1)) = \bot$ by implementation condition 14. Also $ekofdk(t_1) = \bot$. Thus $A_{ekofdk}(\beta(t_1)) = \bot = \beta(ekofdk(t_1))$.

Case 7: *"$F = vkofsk$".*
Similar for the case $F = ekofdk$.

Case 8: *"$F = isenc$".*
If t_1 is of the form $enc(\dots)$, $garbageEnc(\dots)$, or x^c, then $\beta(t_1)$ is of type ciphertext. Remember also that $ekofdk(x^c) = x^c$ (page 27). Thus $A_{isenc}(\beta(t_1)) \overset{(*)}{=} \beta(t_1) = \beta(isenc(t_1))$. Here $(*)$ uses implementation condition 23. If t_1 is not of one of these forms, then $\beta(t_1)$ is

not of type ciphertext, and thus $A_{isenc}(\beta(t_1)) = \bot = \beta(\bot) = \beta(isenc(t_1))$. Here $(*)$ uses implementation condition 23.

Case 9: *"$F = ekof$".*
If $t_1 = enc(ek(u_1), u_2, M)$ with $M \in \mathcal{R}$, we have that $\beta(t_1) = A_{enc}(\beta(ek(u_1)), \beta(u_2), r_M)$. By implementation condition 8, $A_{ekof}(\beta(t_1)) = \beta(ek(u_1))$. Furthermore, $ekof(t_1) = ek(u_1)$, hence $A_{ekof}(\beta(t_1)) = \beta(ekof(t_1))$. If $t_1 = enc(ek(u_1), u_2, N^m)$, by protocol condition 5, t_1 was not honestly generated. Hence, by definition of τ, m is of type ciphertext, and $ek(u_1) = \tau(A_{ekof}(m))$. Thus with Claim 1, $\beta(ek(u_1)) = A_{ekof}(m)$. Furthermore, we have $\beta(t_1) = m$ by definition of β and thus $A_{ekof}(\beta(t_1)) = \beta(ek(u_1)) = \beta(ekof(t_1))$. If $t_1 = garbageEnc(u_1, u_2)$, the proof is analogous. If $t_1 = x^c$, we have that $ekof(t_1) = ek(N^{A_{ekof}(c)})$, and thus $\beta(ekof(t_1)) = A_{ekof}(c) = A_{ekof}(\beta(c))$. In all other cases for t_1, $\beta(t_1)$ is not of type ciphertext, hence $A_{ekof}(\beta(t_1)) = \bot$ by implementation condition 8. Furthermore $ekof(t_1) = \bot$. Thus $\beta(ekof(t_1)) = \bot = A_{ekof}(\beta(t_1))$.

Case 10: *"$F = vkof$".*
If $t_1 = sig(sk(N), u_1, M)$ with $N, M \in \mathcal{N}$, we have that $\beta(t_1) = A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)$. By implementation condition 9, $A_{ekof}(\beta(t_1)) = A_{vk}(r_N)$. Furthermore, $vkof(t_1) = vk(N)$, hence $A_{vkof}(\beta(t_1)) = A_{vk}(r_N) = \beta(vk(N)) = \beta(vkof(t_1))$. If $t_1 = x^c$, then $\beta(vkof(x^c)) = \bot = A_{vkof}(\beta(x^c))$. All other cases for t_1 are handled like in the case of $F = ekof$.

Case 11: *"$F = enc$".*
By protocol condition 1, $t_3 =: N \in \mathcal{N}$. If $t_1 = ek(u_1)$ we have $\beta(\text{eval}_{enc}(t_1, t_2, t_3)) = A_{enc}(\beta(t_1), \beta(t_2), r_N)$ by definition of β. Since $\beta(N) = r_N$, we have $\beta(\text{eval}_{enc}(t_1, t_2, t_3)) = A_{enc}(\beta(t_1), \beta(t_2), \beta(t_3))$. If t_1 is not of the form $ek(u_1)$, then $\text{eval}_{enc}(t_1, t_2, t_3) = \bot$ and by definition of β, $\beta(t_1)$ is not of type encryption key and hence by implementation condition 10, $A_{enc}(\beta(t_1), \dots) = \bot = \beta(enc(t_1, t_2, t_3))$.

Case 12: *"$F = dec$".*
We distinguish the following cases for t_1, t_2:

Case 12.1: *"$t_1 = dk(N)$ and $t_2 = enc(ek(N), u_2, M)$ with $N, M \in \mathcal{R}$".*
Then $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), A_{enc}(A_{ek}(N), \beta(u_2), r_M)) = \beta(u_2)$ by implementation condition 18. Furthermore $\beta(dec(t_1, t_2)) = \beta(u_2)$ by definition of D.

Case 12.2: *"$t_1 = dk(N)$ and $t_2 = enc(ek(N), u_2, N^c)$ with $N \in \mathcal{R}$".*
Then t_2 was produced by τ and hence c is of type ciphertext and $\tau(A_{dec}(A_{dk}(r_N), c)) = u_2$. Then by Claim 1, $A_{dec}(A_{dk}(r_N), c) = \beta(u_2)$ and hence $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), c) = \beta(u_2) = \beta(dec(t_1, t_2))$.

Case 12.3: *"$t_1 = dk(N)$ and $t_2 = enc(u_1, u_2, u_3)$ with $N \in \mathcal{R}$ and $u_1 \neq ek(N)$".*
As shown above (case $F = ekof$), $A_{ekof}(\beta(enc(u_1, u_2, u_3)) = \beta(ekof(enc(u_1, u_2, u_3)) = \beta(u_1)$. Moreover, from Claim 3, $A_{ekof}(\beta(enc(u_1, u_2, u_3)) = \beta(u_1) \neq \beta(ek(N)) = A_{ek}(r_N)$. Thus by implementation condition 11, $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), \beta(enc(u_1, u_2, u_3))) = \bot$. Furthermore, $dec(t_1, t_2) = \bot$ and thus $\beta(dec(t_1, t_2)) = \bot$.

Case 12.4: *"$t_1 = dk(N)$ and $t_2 = garbageEnc(u_1, N^c)$ with $N \in \mathcal{R}$".*
Assume that $m := A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{sk}(r_N), c) \neq \bot$. By implementation condition 11 this implies $A_{ekof}(c) = A_{ek}(r_N)$ and thus $\tau(A_{ekof}(c)) = \tau(A_{ek}(r_N)) = ek(N)$. By protocol condition 5 and construction of τ, t_2 has been not been produced by the protocol. Thus is was produced by τ or red_i. Hence (by construction of τ and red_i) c is of type ciphertext. Then, however, we would have

$\tau(c) = enc(ek(N), \tau(m), N^c) \neq t_2$. This is a contradiction to $t_2 = \tau(c)$, so the assumption that $A_{dec}(\beta(t_1), \beta(t_2)) \neq \bot$ was false. So $A_{dec}(\beta(t_1), \beta(t_2)) = \bot = \beta(\bot) = \beta(dec(t_1, garbageEnc(u_1, N^c)))$.

Case 12.5: *"$t_1 = dk(N^e)$ and $t_2 = garbageEnc(u_1, N^c)$".*
By protocol condition 5 and construction of τ, t_2 has been not been produced by the protocol or τ. Thus is was produced by red_i i.e., $t_2 = red_i(x^c)$.
We consider two cases: First, assume $e = A_{ekof}(c)$. Then by definition of red_i, $red_i(x^c)$ only outputs $garbageEnc(u_1, N^c)$ if $A_{dec}(A^{-1}_{ekofdk}(A_{ekof}(c)), c) = \bot$. Hence $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A^{-1}_{ekofdk}(e), c) = A_{dec}(A^{-1}_{ekofdk}(A_{ekof}(c)), c) = \bot = \beta(\bot) = \beta(dec(t_1, t_2))$.
Second, assume $e \neq A_{ekof}(c)$. Let $d := A^{-1}_{ekofdk}(e)$. Then $A_{ekofdk}(d) \neq A_{ekof}(c)$, and by implementation condition 12, $A_{dec}(d, c) = \bot$. Thus $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A^{-1}_{ekofdk}(e), c) = A_{dec}(d, c) = \bot = \beta(\bot) = \beta(dec(t_1, t_2))$.

Case 12.6: *"$t_1 = dk(N)$ with $N \in \mathbf{N}$ and $t_2 = x^c$ and $A_{ekof}(c) \neq A_{ekofdk}(\beta(t_1))$".*
Since $A_{ekof}(c) \neq A_{ekofdk}(\beta(t_1))$, we have $A_{dec}(\beta(t_1), c) = \bot$ by implementation condition implementation condition 12. Furthermore, $dec(t_1, t_2) = \bot$. Hence $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(\beta(t_1), c) = \bot = \beta(dec(t_1, t_2))$.

Case 12.7: *"$t_1 = dk(N)$ with $N \in \mathbf{N}$ and $t_2 = x^c$ with $A_{ekof}(c) = A_{ekofdk}(\beta(t_1))$".*
This case does not occur. If $N \in \mathcal{R}$, then τ would not output x^c but instead a term $enc(\dots)$ or $garbageEnc(\dots)$. If $N = N^e$ and $A_{ekof}(c) = A_{ekofdk}(\beta(t_1))$, then $A_{ekof}(c) = A_{ekofdk}(\beta(t_1)) = A_{ekofdk}(A^{-1}_{ekofdk}(e)) = e$, and then $red_i(x^c)$ will return $enc(\dots)$ or $garbageEnc(\dots)$. Thus t_2 cannot be x^c.

Case 12.8: *"$t_1 = dk(N^e)$ and $t_2 = enc(ek(N^e), u_2, N^c)$".*
Terms of the form $enc(ek(N^e), u_2, N^c)$ are neither produced by τ (τ only produces enc-terms with encryption keys $ek(N)$, $N \in \mathbf{N}_P$), nor by the protocol (by protocol condition protocol condition 1, the randomness of the enc-term is in $\mathbf{N}_P$). Thus t_2 was produced by red_i, i.e., $t_2 = red_i(x^c)$. But $red_i(x^c)$ only outputs a term $enc(ek(N^e), u_2, N^c)$ if $\tau(A_{dec}(\beta(dk(N^e)), \beta(x^c))) = u_2$. Hence $\beta(dec(t_1, t_2)) = \beta(u_2) = \beta(\tau(A_{dec}(\beta(dk(N^e)), \beta(x^c)))) \overset{(*)}{=} A_{dec}(\beta(dk(N^e)), \beta(x^c)) = A_{dec}(\beta(dk(N^e)), c) = A_{dec}(\beta(t_1), \beta(t_2))$. Here $(*)$ uses Claim 1.

Case 12.9: *"$t_1 = dk(N^e)$ and $t_2 = enc(ek(N^e), u_2, N)$ with $N \in \mathcal{R}$".*
We have $\beta(t_1) = A^{-1}_{ekofdk}(e) =: d$ and hence $\beta(ek(N^e)) = e = A_{ekofdk}(d)$. Then $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(d, A_{enc}(A_{ekofdk}(d), \beta(u_2), r_N)) \overset{(*)}{=} \beta(u_2) = \beta(dec(t_1, t_2))$. Here $(*)$ uses implementation condition 13.

Case 12.10: *"$t_1 = dk(N^e)$ and $t_2 = enc(u_1, u_2, N)$ with $u_1 \neq ek(N^e)$ and $N \in \mathcal{R}$".*
Since $dec(t_1, t_2) = \bot$, we have to show that $A_{dec}(\beta(t_1), \beta(t_2)) = \bot$. We have $\beta(t_1) = A^{-1}_{ekofdk}(e) =: d$. Furthermore, $\beta(u_1) \neq e$ since otherwise we would have $u_1 = ek(N^e)$. Thus $A_{ekofdk}(d) \neq \beta(u_1)$. Then $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(d, A_{enc}(\beta(u_1), \beta(u_2), r_N)) \overset{(*)}{=} \bot$ where $(*)$ uses implementation conditions 12 and 8 and $A_{ekofdk}(d) \neq \beta(u_1)$.

Case 12.11: *"$t_1 = dk(N^e)$ and $t_2 = enc(u_1, u_2, N^c)$ with $u_1 \neq ek(N^e)$".*
Since $dec(t_1, t_2) = \bot$, we have to show that $A_{dec}(\beta(t_1), \beta(t_2)) = \bot$. We have $\beta(t_1) = A^{-1}_{ekofdk}(e) =: d$. Furthermore, since the protocol does not produce terms $enc(\dots, N^c)$, t_2 was produced by τ or red. Thus by definition of τ and red, $u_1 = \tau(A_{ekof}(c))$. Since $\tau(e) = ek(N^e) \neq u_1$, it follows that $A_{ekof}(c) \neq e = A_{ekofdk}(d)$. Thus by implementation condition 12 we have $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(d, c) = \bot$.

Case 12.12: *"t_1 is not of the form $dk(\dots)$".*
Then $\beta(t_1)$ is not of type decryption key. Thus by implementation conditions 12 and 14, $A_{dec}(\beta(t_1), \beta(t_2)) = \bot$. Also, $\beta(dec(t_1, t_2)) = dec(t_1, t_2) = \bot$.

Case 12.13: *"t_2 is not of the form $enc(\dots)$, $garbageEnc(\dots)$, nor x^c".*
Then $\beta(t_2)$ is not of type ciphertext. By implementation condition 8, $A_{ekof}(\beta(t_2)) = \bot$. Hence $A_{ekof}(\beta(t_2)) \neq A_{ek}(r_N)$ and by implementation condition 11, $A_{dec}(\beta(t_1), \beta(t_2)) = A_{dec}(A_{dk}(r_N), \beta(t_2)) = \bot = \beta(dec(t_1, t_2))$.

Case 13: *"$F = sig$".*
By protocol condition 1 we have $t_3 = M \in \mathcal{R}$. If t_1 is of the form $ek(\cdot)$, then $\beta(sig(t_1, t_2, t_3)) = A_{sig}(\beta(t_1), \beta(t_2), r_M) = A_{sig}(\beta(t_1), \beta(t_2), \beta(t_3))$. If $t_1 \neq ek(\cdot)$ then $\beta(sig(t_1, t_2, t_3)) = \bot = A_{sig}(\beta(t_1), \beta(t_2), \beta(t_3))$.

Case 14: *"$F = verify$".*
We distinguish the following subcases:

Case 14.1: *"$t_1 = vk(N)$ and $t_2 = sig(sk(N), u_2, M)$ with $N, M \in \mathcal{N}$".*
Then $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vk}(r_N), A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)) \overset{(*)}{=} \beta(u_2) = \beta(verify(\underline{t}))$ where $(*)$ uses implementation condition 19.

Case 14.2: *"$t_2 = sig(sk(N), u_2, M)$ and $t_1 \neq vk(N)$ with $N, M \in \mathcal{N}$".*
By Claim 3, $\beta(t_1) \neq \beta(vk(N))$ Furthermore $A_{verify}(\beta(vk(N)), \beta(t_2)) = A_{verify}(\beta(t_1), A_{sig}(A_{sk}(r_N), \beta(u_2), r_M)) \overset{(*)}{=} \beta(u_2) \neq \bot$. Hence with implementation condition 20, $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(\bot) = verify(t_1, t_2)$.

Case 14.3: *"$t_1 = vk(N)$ and $t_2 = sig(sk(N), u_2, M^s)$".*
Then t_2 was produced by τ and hence s is of type signature with $\tau(A_{vkof}(s)) = vk(N)$ and $m := A_{verify}(A_{vkof}(s), s) \neq \bot$ and $u_2 = \tau(m)$. Hence with Claim 1 we have $m = \beta(\tau(m)) = \beta(u_2)$ and $\beta(t_1) = \beta(vk(N)) = \beta(\tau(A_{vkof}(s))) = A_{vkof}(s)$. Thus $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vkof}(s), s) = m = \beta(u_2)$. And $\beta(verify(t_1, t_2)) = \beta(verify(vk(N), sig(sk(N), u_2, M^s))) = \beta(u_2)$.

Case 14.4: *"$t_2 = sig(sk(N), u_2, M^s)$ and $t_1 \neq vk(N)$".*
As in the previous case, $A_{verify}(A_{vkof}(s), s) \neq \bot$ and $\beta(vk(N)) = A_{vkof}(s)$. Since $t_1 \neq vk(N)$, by Claim 3, $\beta(t_1) \neq \beta(vk(N)) = A_{vkof}(s)$. From implementation condition 20 and $A_{verify}(A_{vkof}(s), s) \neq \bot$, we have $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(\beta(t_1), s) = \bot = \beta(\bot) = \beta(verify(t_1, t_2))$.

Case 14.5: *"$t_2 = garbageSig(u_1, N^s)$".*
Then t_2 was produced by τ and hence s is of type signature and either $A_{verify}(A_{vkof}(s), s) = \bot$ or $\tau(A_{vkof}(s))$ is not of the form $vk(\dots)$. The latter case only occurs if $A_{vkof}(s) = \bot$ as otherwise $A_{vkof}(s)$ is of type verification key and hence $\tau(A_{vkof}(s)) = vk(\dots)$. Hence in both cases $A_{verify}(A_{vkof}(s), s) = \bot$. If $\beta(t_1) = A_{vkof}(s)$ then $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(A_{vkof}(s), s) = \bot = \beta(verify(t_1, t_2))$. If $\beta(t_1) \neq A_{vkof}(s)$ then by implementation condition 20, $A_{verify}(\beta(t_1), \beta(t_2)) = A_{verify}(\beta(t_1), s) = \bot$. Thus in both cases, with $verify(t_1, t_2) = \bot$ we have $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(verify(t_1, t_2))$.

Case 14.6: *"All other cases".*
Then $\beta(t_2)$ is not of type signature, hence by implementation condition 9, $A_{vkof}(\beta(t_2)) = \bot$, hence $\beta(t_1) \neq A_{vkof}(\beta(t_2))$, and by implementation condition 20 we have $A_{verify}(\beta(t_1), \beta(t_2)) = \bot = \beta(verify(t_1, t_2))$.

Case 15: *"$F = equals$".*

If $t_1 = t_2$ we have $\beta(equals(t_1, t_2)) = \beta(t_1) = A_{equals}(\beta(t_1), \beta(t_1)) = A_{equals}(\beta(t_1), \beta(t_2))$. If $t_1 \neq t_2$, then $t_1, t_2 \notin \mathcal{R}$. To see this, let N_1 be the node associated with t_1. If N_1 is a nonce computation node, then $t_1 \notin \mathcal{R}$ follows from protocol conditions 2, 3, and 4. In case N_1 is an input node, $t_1 \notin \mathcal{R}$ follows by definition of τ. Finally, if N_1 is a destructor computation node, $t_1 \notin \mathcal{R}$ follows inductively. (Similarly for t_2.) By Claim 3, $t_1, t_2 \notin \mathcal{R}$ implies $\beta(t_1) \neq \beta(t_2)$ and hence $\beta(equals(t_1, t_2)) = \bot = A_{equals}(\beta(t_1), \beta(t_2))$ as desired.

Case 16: *"$F \in \{garbage, garbageEnc, garbageSig\}$".*

By protocol condition 5, the constructors *garbage*, *garbageEnc*, and *garbageSig* do not occur in the protocol.

Thus Claim 4 holds.

We will now show that for the random choices fixed above, $Nodes^p_{\mathbf{M},A,\Pi_\mathrm{p},E} = H\text{-}Nodes_{\mathbf{M},\Pi_\mathrm{p},Sim}$.

To prove this, we show the following invariant: $f'_i = \beta \circ red_i \circ f_i^C$ and $\nu'_i = \nu_i^C$ and $s_i = s'_i$ for all $i \geq 0$. We show this by induction on i.

We have $f'_0 = f_0^C = \varnothing$ and $\nu'_0 = \nu_0^C$ is the root node, so the invariant is satisfied for $i = 0$. Assume that the invariant holds for some i. If ν'_i is a nondeterministic node, $\nu'_{i+1} = \nu_{i+1}^C$ is determined by $e_{\nu'_i} = e_{\nu_i^C}$. Since a nondeterministic node does not modify f and the adversary is not activated and $red_i = red_{i+1}$, we have $f'_{i+1} = f'_i = \beta \circ red_i \circ f_i^C = \beta \circ red_{i+1} \circ f_{i+1}^C$ and $s_i = s'_i$. Hence the invariant holds for $i + 1$ if ν'_i is a nondeterministic node.

If ν'_i is a computation node with constructor or destructor F, let $\bar{\nu}_1, \ldots, \bar{\nu}_n$ be the arguments of node ν'_i. Let i_j be the index such that $\bar{\nu}_j = \nu_{i_j}$ (i.e., i_j is the index of the iteration at which node $\bar{\nu}_j$ was processed). By induction hypothesis we have that $f'_{i+1}(\nu'_i) = A_F(f'_i(\bar{\nu}_1), \ldots, f'_i(\bar{\nu}_n)) = A_F(\beta(red_{i_1}(f_i^C(\bar{\nu}_1))), \ldots, \beta(red_{i_n}(f_i^C(\bar{\nu}_n))))$. Since β returns the same bitstring for x^c and $enc(\ldots, N^c)$ and $garbageEnc(\ldots, N^c)$, we have that $\beta(red_{i_j}(f_i^C(\bar{\nu}_j)) = \beta(red_{i+1}(f_i^C(\bar{\nu}_j))$ for $j = 1, \ldots, n$. Hence $f'_{i+1}(\nu'_i) = A_F(\beta(red_{i+1}(f_i^C(\bar{\nu}_1))), \ldots, \beta(red_{i+1}(f_i^C(\bar{\nu}_n))))$. And $f_{i+1}^C(\nu'_i) = f_{i+1}^C(\nu_i^C) = F(f_i^C(\bar{\nu}_1), \ldots, f_i^C(\bar{\nu}_n))$. From Claim 4 it follows that $\beta(red_{i+1}(f_{i+1}^C(\nu'_i))) = \beta(\mathrm{eval}_F(red_{i+1}(f_i^C(\bar{\nu}_1)), \ldots, red_{i+1}(f_i^C(\bar{\nu}_n)))) = f'_{i+1}(\nu'_i)$ where the lhs is defined iff the rhs is. Hence $\beta \circ red_{i+1} \circ f_{i+1}^C = f'_{i+1}$.

In the hybrid execution, the yes-successor is taken iff the simulator answers yes to the question $Q := F(f_i^C(\bar{\nu}_1), \ldots, f_i^C(\bar{\nu}_n)) = f_{i+1}^C(\nu_i)$. The simulator answers yes iff $red_{i+1}(f_{i+1}^C(\nu_i)) = red_{i+1}(Q) \neq \bot$. By Claim 2, $red_{i+1}(f_{i+1}^C(\nu_i)) \neq \bot$ iff $\beta(red_{i+1}(f_{i+1}^C(\nu_i))) \neq \bot$. And as shown above, this holds iff $f'_{i+1}(\nu_i)$. And in the computational execution, the yes-success is taken iff $A_F(f'_i(\bar{\nu}_1), \ldots, f'_i(\bar{\nu}_n)) = f'_{i+1}(\nu_i) \neq \bot$. Thus the yes-successor is taken in the hybrid execution iff it is taken in the computational execution. Thus $\nu_{i+1}^C = \nu'_{i+1}$.

The adversary E is not invoked, hence $s'_{i+1} = s_{i+1}^C$. So the invariant holds for $i + 1$ if ν'_i is a computation node with a constructor or destructor.

If ν'_i is a computation node with nonce $N \in \mathbf{N}_P$, we have that $f'_{i+1}(\nu'_i) = r_N = \beta(N) = \beta(red_{i+1}(f_{i+1}^C(\nu'_i)))$. Hence $\beta \circ red_i \circ f_{i+1}^C = f'_{i+1}$. Since $A_N() \neq \bot$, ν'_{i+1} is the yes-successor of ν'_i. Since $red_i(N) = N \neq \bot$ the simulator answers yes to the question $Q := N$, and thus ν_{i+1}^C is the yes-successor of $\nu_i^C = \nu'_i$. Thus $\nu'_{i+1} = \nu_{i+1}^C$. The adversary E is not invoked, hence $s'_{i+1} = s_{i+1}^C$. So the invariant holds for $i + 1$ if ν'_i is a computation node with a nonce.

In the case of a control node, the adversary E in the computational execution and the simulator in the hybrid execution get the out-metadata l of the node ν'_i or ν_i^C, respectively. The simulator passes l on to the simulated adversary. Thus, since $s'_i = s_i^C$, we have that $s'_{i+1} = s_{i+1}^C$, and in the computational and the hybrid execution, E answer with the same in-metadata l'. Thus $\nu'_{i+1} = \nu_{i+1}^C$. Since a control node does not modify f and $red_i = red_{i+1}$ we

have $f'_{i+1} = f'_i = \beta \circ red_i \circ f_i^C = \beta \circ red_{i+1} \circ f_{i+1}^C$. Hence the invariant holds for $i+1$ if ν'_i is a control node.

In the case of an input node, the adversary E in the computational execution and the simulator in the hybrid execution is asked for a bitstring m' or term t^C, respectively. The simulator produces this string by asking the simulated adversary E for a bitstring m^C and setting $t^C := \tau(m^C)$. Since $s'_i = s_i^C$, $m' = m^C$. Then by definition of the computational and hybrid executions, $f'_{i+1}(\nu'_i) = m'$ and $f_{i+1}^C(\nu'_i) = t^C = \tau(m')$. Thus $f'_{i+1}(\nu'_i) = m' \overset{(*)}{=} \beta(\tau(m')) = \beta(red_{i+1}(\tau(m))) = \beta(red_{i+1}(f_{i+1}^C(\nu'_i)))$ where $(*)$ follows from Claim 1. Since $f'_{i+1} = f'_i$ and $f_{i+1}^C = f^C$ everywhere else, we have $f'_{i+1} = \beta \circ red_{i+1} \circ f_{i+1}^C$. Furthermore, since input nodes have only one successor, $\nu'_{i+1} = \nu_{i+1}^C$. Thus the invariant holds for $i+1$ in the case of an input node.

In the case of an output node, the adversary E in the computational execution gets $m' := f'_i(\bar{\nu}_1)$ where the node $\bar{\nu}_1$ depends on the label of ν'_i. In the hybrid execution, the simulator gets $t^C := f_i^C(\bar{\nu}_1)$ and sends $m^C := \beta(red_i(t^C)) = \beta(red_{i+1}(t^C))$ to the simulated adversary E. By induction hypothesis we then have $m' = m^C$, so the adversary gets the same input in both executions. Thus $s'_{i+1} = s_{i+1}^C$. Furthermore, since output nodes have only one successor, $\nu'_{i+1} = \nu_{i+1}^C$. And $f'_{i+1} = f'_i$ and $f_{i+1}^C = f^C$, so $f'_{i+1} = \beta \circ red_{i+1} \circ f_{i+1}^C$. Thus the invariant holds for $i+1$ in the case of an output node.

From the invariant it follows that the node trace is the same in both executions.

Since random choices with all nonces, keys, encryptions, and signatures being pairwise distinct occur with overwhelming probability (as mentioned at the beginning of this proof), the node traces of the real and the hybrid execution are indistinguishable. □

Lemma 6 *In a given step of the hybrid execution with Sim_7, let S be the set of messages sent from Π^c to Sim_7 up to that step. Let $u' \in \mathbf{T}_x$ be the message sent from Sim_7 to Π^c in that step. Let φ denote the final substitution output by Sim_7. Let $\mathcal{C}$ be a context and $u \in \mathbf{T}_x$ such that $u'\varphi = \mathcal{C}[u]$ and $S\varphi \nvdash u$ and $\mathcal{C}$ does not contain a subterm of the form $sig(\Box, \cdot, \cdot)$. ($\Box$ denotes the hole of the context $\mathcal{C}$.)*

Then there exists a term t_{bad} and a context $\mathcal{D}$ such that $\mathcal{D}$ obeys the following grammar

$$\begin{aligned}\mathcal{D} ::= \Box \mid & \mathit{pair}(t, \mathcal{D}) \mid \mathit{pair}(\mathcal{D}, t) \mid \mathit{enc}(\mathit{ek}(N), \mathcal{D}, M) \mid \mathit{enc}(\mathit{ek}(M), \mathcal{D}, M) \\ & \mid \mathit{enc}(\mathcal{D}, t, M) \mid \mathit{sig}(\mathit{sk}(M), \mathcal{D}, M) \mid \mathit{sig}(\mathit{sk}(N), \mathcal{D}, M) \\ & \mid \mathit{garbageEnc}(\mathcal{D}, M) \mid \mathit{garbageSig}(\mathcal{D}, M) \\ & \quad \text{with } N \in \mathbf{N}_P, M \in \mathbf{N}_E, t \in \mathbf{T}\end{aligned}$$

and such that $u = \mathcal{D}[t_{bad}]$ and such that $S\varphi \nvdash t_{bad}$ and such that one of the following holds:

- $t_{bad} \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{enc}(p, m, N)$ *with* $N \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{sig}(k, m, N)$ *with* $N \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{sig}(\mathit{sk}(N), m, M)$ *with* $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ *and* $S\varphi \nvdash \mathit{sk}(N)$ *or*
- $t_{bad} = \mathit{ek}(N)$ *with* $N \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{vk}(N)$ *with* $N \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{dk}(N)$ *with* $N \in \mathbf{N}_P$, *or*
- $t_{bad} = \mathit{sk}(N)$ *with* $N \in \mathbf{N}_P$.

Proof. We prove the lemma by structural induction on u. We distinguish the following cases:

Case 1: *"$u = \mathit{garbage}(u_1)$".*

By protocol condition 5 the protocol does not contain *garbage*-computation nodes. Thus u

is not an honestly generated term. Hence it was produced by an invocation $\tau(m)$ for some $m \in \{0,1\}^*$, and hence $u = \mathit{garbage}(N^m)$. Hence $S\varphi \vdash u$ in contradiction to the premise of the lemma.

Case 2: *"$u = \mathit{garbageEnc}(u_1, u_2)$".*
By protocol condition 5 the protocol does not contain *garbageEnc*-computation nodes. Thus u is not an honestly generated term. Hence u was produced by $\tau(c)$ or $\mathit{red}_i(x^c)$ for some c, and hence $u = \mathit{garbageEnc}(u_1, N^c)$. Since $S\varphi \vdash N^c$ and $S\varphi \nvdash u$, we have $S\varphi \nvdash u_1$. Hence by the induction hypothesis, there exists a subterm t_{bad} of u_1 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_1. Then t_{bad} and $\mathcal{D}' := \mathit{garbageEnc}(\mathcal{D}, N^c)$ satisfy the conclusion of the lemma for u.

Case 3: *"$u = \mathit{garbageSig}(u_1, u_2)$".*
By protocol condition 5 the protocol does not contain *garbageSig*-computation nodes. Thus u is not an honestly generated term. Hence it was produced by an invocation $\tau(c)$ for some $c \in \{0,1\}^*$, and hence $u = \mathit{garbageSig}(u_1, N^m)$. Since $S\varphi \vdash N^m$ and $S\varphi \nvdash u$, we have $S\varphi \nvdash u_1$. Hence by the induction hypothesis, there exists a subterm t_{bad} of u_1 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_1. Then t_{bad} and $\mathcal{D}' := \mathit{garbageSig}(\mathcal{D}, N^m)$ satisfy the conclusion of the lemma for u.

Case 4: *"$u \in \{ek(u_1), vk(u_1), dk(u_1), sk(u_1)\}$ with $u_1 \notin \mathbf{N}_P$".*
By protocol condition 1, the argument of an ek-, $vk(u_1)$, $dk(u_1)$-, or $sk(u_1)$-computation node is an N-computation node with $N \in \mathbf{N}_P$. Hence u is not honestly generated. Hence it was produced by an invocation of τ, and hence $u \in \{ek(N^e), vk(N^e), dk(N^e), sk(N^e)\}$ for some e. Hence $S\varphi \vdash u$ in contradiction to the premise of the lemma.

Case 5: *"$u \in \{ek(N), vk(N), dk(N), sk(N)\}$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 6: *"$u = \mathit{pair}(u_1, u_2)$".*
Since $S\varphi \nvdash u$, we have $S\varphi \nvdash u_i$ for some $i \in \{1,2\}$. Hence by induction hypothesis, there exists a subterm t_{bad} of u_i and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_i. Then t_{bad} and $\mathcal{D}' = \mathit{pair}(\mathcal{D}, u_2)$ or $\mathcal{D}' = \mathit{pair}(u_1, \mathcal{D})$ satisfy the conclusion of the lemma for u.

Case 7: *"$u = \mathit{string}_i(u_1)$ with $i \in \{0,1\}$ or $u = \mathit{empty}$".*
Then, since $u \in \mathbf{T}$, u contains only the constructors $\mathit{string}_0, \mathit{string}_1, \mathit{empty}$. Hence $S\varphi \vdash u$ in contradiction to the premise of the lemma.

Case 8: *"$u \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 9: *"$u \in \mathbf{N}_E$".*
Then $S\varphi \vdash u$ in contradiction to the premise of the lemma.

Case 10: *"$u = \mathit{enc}(u_1, u_2, N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 11: *"$u = \mathit{enc}(u_1, u_2, u_3)$ with $S\varphi \nvdash u_1$ and $u_3 \notin \mathbf{N}_P$".*
Since $u_1 \notin \mathbf{N}_P$, $u = \mathit{enc}(u_1, u_2, N^c)$ for some c. Since $S\varphi \nvdash u_1$, by induction hypothesis, there exists a subterm t_{bad} of u_1 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_1. Then t_{bad} and $\mathcal{D}' = \mathit{enc}(\mathcal{D}, u_2, N^c)$ satisfy the conclusion of the lemma for u.

Case 12: *"$u = enc(u_1, u_2, u_3)$ with $S\varphi \vdash u_1$ and $u_3 \notin \mathbf{N}_P$".*
Since $u_1 \notin \mathbf{N}_P$, $u = enc(ek(N), u_2, N^c)$ for some c, N with $N \in \mathbf{N}_P$ or $N \in \mathbf{N}_E$. From $S\varphi \vdash u_1$, $S\varphi \vdash N^c$, and $S\varphi \nvdash u$ we have $S\varphi \nvdash u_2$. Hence by induction hyposthesis, there exists a subterm t_{bad} of u_2 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_2. Then t_{bad} and $\mathcal{D}' = enc(ek(N), \mathcal{D}, N^c)$ satisfy the conclusion of the lemma for u.

Case 13: *"$u = sig(u_1, u_2, N)$ with $N \in \mathbf{N}_P$".*
The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 14: *"$u = sig(sk(N), u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $N \in \mathbf{N}_P$ and $S\varphi \nvdash sk(N)$".*
Since $u \in \mathbf{T}$ we have $u_3 \in \mathbf{N}$, hence $u_3 \in \mathbf{N}_E$. The conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 15: *"$u = sig(sk(N), u_2, u_3)$ with with $u_3 \notin \mathbf{N}_P$ and $N \in \mathbf{N}_P$ and $S\varphi \vdash sk(N)$".*
Since $u \in \mathbf{T}$ we have $u_3 \in \mathbf{N}$, hence $u_3 \in \mathbf{N}_E$ and thus $S\varphi \vdash u_3$. Since $S\varphi \nvdash u$ but $S\varphi \vdash sk(N)$ and $S\varphi \vdash u_3$, we have $S\varphi \nvdash u_2$. Hence by induction hyposthesis, there exists a subterm t_{bad} of u_2 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_2. Then t_{bad} and $\mathcal{D}' = sig(sk(N), \mathcal{D}, u_3)$ satisfy the conclusion of the lemma for u.

Case 16: *"$u = sig(u_1, u_2, u_3)$ with $S\varphi \vdash u_1$ and $u_3 \notin \mathbf{N}_P$ and u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$".*
By protocol condition 1, the third argument of an E-computation node is an N-computation node with $N \in \mathbf{N}_P$. Hence u is not honestly generated. Hence it was produced by an invocation $\tau(s)$ for some $s \in \{0,1\}^*$, and hence $u = sig(sk(M), u_2, N^s)$ for some $M \in \mathbf{N}$. Since u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$, we have $M \in \mathbf{N}_E$. From $S\varphi \vdash u_1$, $S\varphi \vdash N^c$, and $S\varphi \nvdash u$ we have $S\varphi \nvdash u_2$. Hence by induction hyposthesis, there exists a subterm t_{bad} of u_2 and a context $\mathcal{D}$ satisfying the conclusion of the lemma for u_2. Then t_{bad} and $\mathcal{D}' = sig(sk(M), \mathcal{D}, N^s)$ satisfy the conclusion of the lemma for u.

Case 17: *"$u = sig(u_1, u_2, u_3)$ with $S\varphi \nvdash u_1$ and $u_3 \notin \mathbf{N}_P$".*
As in the previous case, $u = sig(sk(N), u_2, N^s)$ for some $N \in \mathbf{N}$. Since $S\varphi \nvdash u_1$, $N \notin \mathbf{N}_E$. Hence $N \in \mathbf{N}_P$. Thus conclusion of the lemma is fulfilled with $\mathcal{D} := \Box$ and $t_{bad} := u$.

Case 18: *"$u = x^c$".*
We have $u'\varphi = \mathcal{C}[u]$. Furthermore, since φ maps all x^c to terms in $\mathbf{T}$ (i.e., without variables), it follows that u cannot contain x^c. So this case does not occur.

□

Lemma 7 *Sim_7 is DY for $\mathbf{M}$ and Π.*

Proof. If Sim_7 is not DY, then with not-negligible probability in the hybrid execution of Sim_7, we have an invocation $u' = \tau(m')$ such that $S\varphi \nvdash u'\varphi$ where S is the set of messages sent by the protocol to Sim_7 up to that invocation.

We introduce the following notation: $t_1 \leq^{\mathcal{D}} t_2$ means that $t_2 = \mathcal{D}[t_1]$ for some context $\mathcal{D}$ matching the grammar from Lemma 6. We write $t_1 \leq^{\mathcal{D}}_{\varphi} t_2$ if there is a a t_1^* with $t_1^*\varphi = t_1$ and $t_1^* \leq^{\mathcal{D}} t_2$. We say t_{bad} is a unpredicability-atom, if it falls in one of the eight cases listed in Lemma 6.

By Lemma 6 (with $u := u'\varphi$ and $\mathcal{C} := \Box$), there exists a unpredicability-atom $t_{bad} \leq^{\mathcal{D}} u'\varphi$ with $S\varphi \nvdash t_{bad}$. Without loss of generality, let u' be the first output of τ that contains such a t_{bad}.

Since $t_{bad} \leq^{\mathcal{D}} u'\varphi$, we distinguish two cases. First, t_{bad} corresponds to some subterm of u', i.e., $t_{bad} \leq^{\mathcal{D}}_{\varphi} u'$. Second, $t_{bad} \leq^{\mathcal{D}} \varphi(x^c)$ for some x^c occurring in u'.

Consider the second case first: Since by construction, $\varphi(x^c)$ is of the form $enc(N^e, t', N^c)$ or $garbageEnc(ek(N^e), N^c)$, we have that $\varphi(x^c)$ is not a random-atom, hence t_{bad} is a proper subterm of $\varphi(x^c)$. Furthermore, since $\not\vdash garbageEnc(ek(N^e), N^c)$, $\varphi(x^c)$ must be of the form $enc(N^e, t', N^c)$ and $t_{bad} \leq^{\mathcal{D}} t'$ and $S\varphi \not\vdash t'$. In this case, by construction of φ, $t' = \tau(A_{dec}(d, c))\varphi$ where $d := A^{-1}_{ekofdk}(e)$. Remember that Sim_7 aborts with a malicious-key-extraction-failure if $A_{dec}(d, c)$ returns a value $\tilde{m}$ that was not output by the malicious-key-extractor MKE when x^c first occurred. Thus $t_{bad} \leq^{\mathcal{D}} \tau(\tilde{m})\varphi$. But $\tau(\tilde{m})$ was invoked before $\tau(m')$, this contradicts the assumption that u' be the first output of τ that contains a term t_{bad}.

Thus the second case cannot occur, and we have $t_{bad} \leq^{\mathcal{D}}_{\varphi} u'$. Thus there is a term t^*_{bad} such that $t^*_{bad}\varphi = t_{bad}$ and $u = \mathcal{D}[t^*_{bad}]$ for some $\mathcal{D}$ matching the grammar from Lemma 6. And t^*_{bad} is also a unpredicability-atom (if it was not, then $t_{bad} = t^*_{bad}$ cannot be a unpredicability-atom either, since the images $\varphi(x^c)$ are no unpredicability-atoms).

Since $S\varphi \not\vdash t_{bad} = t'_{bad}\varphi$, we have by Lemma 1 that $\beta^\dagger(t'_{bad})$ is not invoked the invocation in which $u' = \tau(m')$ was invoked.

By definition of τ and by the syntax of $\mathcal{D}$, during the computation of $\tau(m') = u'$, we have $\tau(m) = t'_{bad}$ for some recursive invocation $\tau(m)$ of τ. Hence the simulator has computed a bitstring m_{bad} with $\tau(m_{bad}) = t'_{bad}$.

We are left to show that such a bitstring m_{bad} can be found only with negligible probability.

We distinguish the possible values for the unpredicability-atom t'_{bad} (as listed in Lemma 6):

Case 1: *"$t'_{bad} = N \in \mathbf{N}_P \setminus \mathcal{R}$".*
By construction, Sim_7 accesses r_N only when computing $\beta^\dagger(N)$ and in τ. Since $S \not\vdash t'_{bad} = N$ we have that $\beta^\dagger(N)$ is never invoked, thus r_N is never accessed through $\beta^\dagger$. In τ, r_N is only used in comparisons. More precisely, $\tau(r)$ checks for all $N \in \mathcal{N}$ whether $r = r_N$. Such checks do not help in guessing r_N since when such a check succeeds, r_N has already been guessed. Thus the probability that $m_{bad} = r_N$ occurs as input of τ is negligible.

Case 2: *"$t'_{bad} = N \in \mathcal{R}$".*
This case does not occur since τ only outputs nonces in $\mathbf{N} \setminus \mathcal{R}$.

Case 3: *"$t'_{bad} = enc(p, m, N)$ with $N \in \mathbf{N}_P$".*
Then $\tau(m_{bad})$ returns t'_{bad} only if m_{bad} was the output of an invocation of $\beta^\dagger(enc(p, m, N)) = \beta^\dagger(t'_{bad})$. But $\beta^\dagger(t'_{bad})$ is never invoked, so this case does not occur.

Case 4: *"$t'_{bad} = sig(k, m, N)$ with $N \in \mathbf{N}_P$".*
Then $\tau(m_{bad})$ returns t'_{bad} only if m_{bad} was the output of an invocation of $\beta^\dagger(sig(k, m, N)) = \beta^\dagger(t'_{bad})$. But by Lemma 1, $\beta^\dagger(t'_{bad})$ is never invoked, so this case does not occur.

Case 5: *"$t'_{bad} = sig(sk(N), m, M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ and $S\varphi \not\vdash sk(N)$".*
Then $\tau(m_{bad})$ returns t'_{bad} only if m_{bad} was not the output of an invocation of $\beta^\dagger$. In particular, m_{bad} was not produced by the signing oracle. Furthermore, $\tau(m_{bad})$ returns t'_{bad} only if m_{bad} is a valid signature with respect to the verification key vk_N. Hence m_{bad} is a valid signature that was not produced by the signing oracle. Furthermore, since $S\varphi \not\vdash sk(N)$, by Lemma 1, $\beta^\dagger(t'_{bad})$ was never invoked. Thus the secret key sk_N was never queried from the signing oracle. Hence such a bitstring m_{bad} can only be produced with negligible probability by the adversary E because of the strong existential unforgeability of $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ (implementation condition 27).

Case 6: *"$t'_{bad} = ek(N)$ with $N \in \mathbf{N}_P$".*
Then $\beta^\dagger(ek(N))$ is never computed and hence ek_N never requested from the ciphertext simulator. Since $S \not\vdash ek(N)$, we have $S \not\vdash dk(N)$. Hence by Lemma 1, $\beta^\dagger(dk(N))$ is never computed and dk_N is never requested from the ciphertext simulator. Furthermore, since $S \not\vdash ek(N)$, for all terms of the form $t = enc(ek(N), \dots, \dots)$, we have that $S \not\vdash t$. Thus $\beta^\dagger(t)$ is never computed and hence no encryption using ek_N is ever requested from the ciphertext simulator. However, decryption queries with respect to dk_N may still be sent to the ciphertext simulator. Yet, by implementation condition 11, these will always fail unless the ciphertext to be decrypted already satisfies $A_{ekof}(m) = ek_N$, i.e., if ek_N has already been guessed. Hence the probability that $ek_N = m_{bad}$ occurs as input of τ is negligible.

Case 7: *"$t'_{bad} = vk(N)$ with $N \in \mathbf{N}_P$".*
Then $\beta^\dagger(vk(N))$ is never computed and hence vk_N is never requested from the signing oracle. Furthermore, since $S \not\vdash vk(N)$, we also have $S \not\vdash sk(N)$ and $S \not\vdash t$ for $t = sig(sk(N), \dots, \dots)$. Thus $\beta^\dagger(sk(N))$ and $\beta^\dagger(t)$ never computed and hence neither sk_N nor a signature with respect to sk_N is requested from the signing oracle, i.e., the signing oracle is never queried at all. Hence the probability that $vk_N = m_{bad}$ occurs as output of τ is negligible.

Case 8: *"$t'_{bad} = dk(N)$ with $N \in \mathbf{N}_P$".*
Then $\beta^\dagger * (dk(N))$ is never invoked. Thus dk_N is not queried from the ciphertext simulator. Being able to guess dk_N without quering in from the ciphertext simulator would contradict PROG-KDM security.

Case 9: *"$t'_{bad} = sk(N)$ with $N \in \mathbf{N}_P$".*
Then $\beta^\dagger(sk(N))$ is never invoked. Thus sk_N is never queried from the signing oracle. From the strong existential unforgeability of $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ (implementation condition 27) it follows that the probability of guessing $m_{bad} = sk_N$ without querying sk_N from the signing oracle is negligible.

Summarizing, we have shown that if the simulator Sim_7 is not DY, then with not-negligible probability Sim_7 performs the computation $\tau(m_{bad})$, but m_{bad} can only occur with negligible probability as an argument of τ. Hence we have a contradiction to the assumption that Sim_7 is not DY. □

We are now ready to prove the main theorem, Theorem 1.

Proof of Theorem 1. We use Theorem 1 from [5]. According to that theorem, we need to show that Sim satisfies the following four conditions (for formal definitions see [5]):

- 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.

By Lemma 7, Sim_7 is DY for randomness-safe protocols. Since the full traces of the hybrid execution of Sim_7 and Sim are computationally indistinguishable (Lemma 2), it follows that Sim is DY for randomness-safe protocols.

By Lemma 5, *Sim* is indistinguishable.
By Lemma 4, *Sim* is consistent.
Finally, *Sim* is abort-free by construction (we have no abort-instruction in the definition of *Sim*).

Thus Theorem 1 from [5] applies, and it follows that the implementation A is computationally sound. □

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 restricting the protocol. In *ACM CCS 2012*. ACM Press, October 2012. To appear.

[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.

Symbol index

η	Security parameter
ek	Encryption key
dk	Decryption key
$x \leftarrow A$	Pick x according to algorithm/distribution A

$x \xleftarrow{\$} S$	Pick x uniformly from set S	
reg_R	Register R (in real/fake challenger)	
$\mathcal{O}$	Denotes some oracle (usually the random oracle)	4
$\|x\|$	Absolute value of x	
$\mathcal{A}$	Usually denotes the adversary	
$\mathbb{N}$	Set of natural numbers $1, 2, \ldots$	4
eval_f	Application of constructor or destructor f	
RC	Real challenger	11
$cipher_N$	Set of ciphertexts produced in real/fake challenger in session N	11
$R := \texttt{getek}_{\texttt{ch}}(N)$	Real/fake challenger query: Load encryption key (session N)	11
$R := \texttt{getdk}_{\texttt{ch}}(N)$	Real/fake challenger query: Load decryption key (session N)	11
$R := \texttt{eval}_{\texttt{ch}}(C, R_1, \ldots, R_n)$	Real/fake challenger query: Evaluate circuit C	11
$R := \texttt{enc}_{\texttt{ch}}(N, R_1)$	Real/fake challenger query: Encrypt reg_{R_1} (session N)	11
$\texttt{oracle}_{\texttt{ch}}(x)$	Real/fake challenger query: Oracle query	11
$\texttt{dec}_{\texttt{ch}}(N, c)$	Real/fake challenger query: Decrypt c (session N)	11
`forbidden`	Output denoting attempted decryption of challenge ciphertext	11
$\texttt{reveal}_{\texttt{ch}}(R_1)$	Query type of the real/fake challenger	11
$\texttt{fakeenc}_{\texttt{cs}}(R, l)$	Ciphertext simulator query: Produce fake encrytion	12
$\texttt{dec}_{\texttt{cs}}(c)$	Ciphertext simulator query: Decrypt	12
$\texttt{enc}_{\texttt{cs}}(R, m)$	Ciphertext simulator query: Encryption (non-fake)	12
$\texttt{getek}_{\texttt{cs}}()$	Ciphertext simulator query: Get encryption key	12
$\texttt{getdk}_{\texttt{cs}}()$	Ciphertext simulator query: Get decryption key	12
$\texttt{program}_{\texttt{cs}}(R, m)$	Ciphertext simulator query: Program the oracle	12
CS	Ciphertext simulator	12
FC	Fake challenger	12
`FCRetrieve`	Retrieve function of FC	13
$plain_N$	Plaintexts of fake encryptions from CS_N (part of state of `FCRetrieve`)	13
$\ell_{ek}(\eta)$	Length of an encryption key	13
$\ell_{dk}(\eta)$	Length of a decryption key	13
$\ell_c(\eta, l)$	Length of the encryption of a ciphertext of length l	13
`FCLen`	Length function of FC	13
MKE	Malicious key extractor	18
KeyGen	Key generation algorithm corresponding to A_{ek}, A_{dk}	26
Enc	Encryption algorithm corresponding to A_{enc}	26
Dec	Decryption algorithm corresponding to A_{dec}	26
Sig	Signing algorithm corresponding to A_{sig}	26
Verify	Signature verification algorithm corresponding to A_{verify}	26
SKeyGen	Key generation algorithm corresponding to A_{vk}, A_{sk}	26
$\mathbf{T}_x$	Well-formed terms containing variables	27
$T_{\mathbf{D},x}$	Destructor terms (containing also variables)	27
eval	Recursive evaluation of destructor term	27
Sim	Simulator (unmodified)	27
Sim_1	Simulator (using fresh randomness for encryption)	30
Sim_2	Simulator (using the real challenger)	30

Sim_3	Simulator (using the malicious-key extractor)	31
Sim_4	Simulator (using the fake challenger)	31
Sim_5	Simulator (using the ciphertext simulator)	31
Sim_6	Simulator (using fresh randomness for signing)	32
Sim_7	Simulator (using a signing oracle)	32
$\mathcal{O}_N^{\mathsf{sig}}$	Signing oracle for nonce N	32

Index