Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library

Michael Backes, Birgit Pfitzmann
IBM Zurich Research Lab

Abstract

Recently we solved the long-standing open problem of justifying a Dolev-Yao type model of cryptography as used in virtually all automated protocol provers under active attacks. The justification was done by defining an ideal system handling Dolev-Yao-style terms and a cryptographic realization with the same user interface, and by showing that the realization is as secure as the ideal system in the sense of reactive simulatability. This definition encompasses arbitrary active attacks and enjoys general composition and property-preservation properties. Security holds in the standard model of cryptography and under standard assumptions of adaptively secure primitives.

A major primitive missing in that library so far is symmetric encryption. We show why symmetric encryption is harder to idealize in a way that allows general composition than existing primitives in this library. We discuss several approaches to overcome these problems. For our favorite approach we provide a detailed provably secure idealization of symmetric encryption within the given framework for constructing nested terms.

1 Introduction

Automated proofs of security protocols with model checkers or theorem provers typically abstract from cryptography by deterministic operations on abstract terms and by simple cancellation rules. An example term is $\mathsf{E}_{pke_w}(\mathsf{E}_{pke_v}(\mathsf{sign}_{sks_u}(m, N_1), N_2))$, where m denotes an application message and N_1, N_2 two nonces. A typical cancellation rule is $\mathsf{D}_{ske}(\mathsf{E}_{pke}(m)) = m$ for corresponding keys. The proof tools handle these terms symbolically, i.e., they never evaluate them to bitstrings. In other words, they perform abstract algebraic manipulations on trees consisting of operators and base messages, using the cancellation rules, the transition rules of a particular protocol, and abstract models of networks and adversaries. Such abstractions, although different in details, are called the Dolev-Yao model after the first authors [17].

For many years there was no cryptographic justification for such abstractions. The problem lies in the assumption, implicit in the adversary model, that actions that cannot be expressed with the abstract operations are impossible, and that no relations hold between terms unless derivable by the cancellation rules. It is not hard to make artificial counterexamples to these assumptions. Nevertheless, no counterexamples against the method for protocols proved in the literature were found so far. Further, the overall approach of abstracting from cryptographic primitives once with rigorous hand-proofs, and then using tools for proving protocols using such primitives, is highly attractive: Besides the cryptographic aspects, protocol proofs have many distributed-systems aspects, which make proofs tedious and error-prone even if they weren't interlinked with the cryptographic aspects. To use existing efficient automated proof tools for security protocols, cryptography must indeed be abstracted into simple, deterministic ideal systems. The closer one can stay to the Dolev-Yao model, the easier the adaptation of the proof tools will be.[1]

Cryptographic underpinnings of a Dolev-Yao model were first addressed by Abadi and Rogaway in [2]. However, they only handled passive adversaries and symmetric encryption. The protocol language and security properties handled were extended in [1, 20], but still only for passive adversaries. This excludes

[1] Efforts are also under way to formulate syntactic calculi for dealing with probabilism and polynomial-time considerations, in particular [23, 21, 24, 18] and, as a second step, to encode them into proof tools. However, this approach can not yet handle protocols with any degree of automation. Generally it is complementary to, rather than competing with, the approach of proving simple deterministic abstractions of cryptography and working with those wherever cryptography is only used in a blackbox way.

most of the typical ways of attacking protocols, e.g., man-in-the-middle attacks and attacks by reusing a message part in a different place or a concurrent protocol run. A full cryptographic justification for a Dolev-Yao model, i.e., for arbitrary active attacks and within arbitrary surrounding interactive protocols, was first given recently in [4]. Based on the specific Dolev-Yao model whose soundness was proven in [4], the well-known Needham-Schroeder-Lowe protocol was proved in [3]. This shows that in spite of adding certain operators and rules compared with simpler Dolev-Yao models (in order to be able to use arbitrary cryptographically secure primitives without too many changes in the cryptographic realization), such a proof is possible in the style already used in automated tools, only now with a sound cryptographic basis. In [5] it was shown how the library, in other words the term algebra and rules, can be modularly extended by additional cryptographic primitives, using the example of symmetric authentication [5].

Nevertheless, symmetric encryption is still missing in this framework, while it is the most common cryptographic primitive in typical proofs with Dolev-Yao models. The goal of this paper is to add symmetric encryption to this framework. Concurrently to our work, Laud [19] has presented a cryptographic underpinning for a Dolev-Yao model of symmetric encryption under active attacks. His work enjoys a direct connection with a formal proof tool, but it is specific to certain confidentiality properties, restricts the surrounding protocols to straight-line programs in a specific language, and does not address a connection to the remaining primitives of the Dolev-Yao model.

There are intrinsic difficulties in providing a sound abstraction from symmetric encryption in the strong sense of security used in [4]. This strong notion is the concept of simulatability. Essentially, it is the cryptographic notion of secure implementation. Very roughly, a real system is called as secure as an ideal system in this sense if everything that can happen to honest users of the real system can also happen to the same honest users with the ideal system. This is typically proved by providing a simulator that, interacting with the ideal system and the honest users, and using an adversary on the real system as a blackbox subsystem, simulates all visible actions of the real system online (i.e., at the time they occur).

For symmetric encryption, there is the following so-called *commitment problem* if one wants to achieve simulatability.[2] The ideal encryption system must somehow allow that secret keys are sent from one participant to another, because many protocols to be proven using such an ideal system are key-exchange protocols. This is the main difference to public-key systems, where an ideal system can assume that only public keys are sent around, because this is sufficient for all standard protocols. If the ideal system simply allows keys to be sent at any time (and typical Dolev-Yao models do allow all valid terms to be sent at any time), the following problem can occur: An honest participant first sends a ciphertext such that the adversary can see it, and later sends both the contained cleartext and the key. This behavior may even be reasonably designed into protocols, e.g., the ciphertext might be an encrypted bet that is later opened. The simulator will first learn in some abstract way that a ciphertext was sent and has to simulate it by some bitstring, which the adversary sees. Later the simulator sees abstractly that a key becomes known and that the ciphertext contains a specific application message. It cannot change the application message, thus it must simulate a key that decrypts the old ciphertext bitstring (produced without knowledge of the application message) to this specific message.

We discuss several ways of dealing with this problem. Our preferred one, for which we actually present the ideal and real symmetric encryption system, is to leave it to the surrounding protocol to guarantee that the commitment problem does not occur. Essentially, this means that the surrounding protocol must guarantee that keys are no longer sent in a form that might make them known to the adversary once an honest participant has started using them. Alternatives would be to build such a guarantee into the ideal and the real system, or to restrict oneself to the few encryption systems where this problem does not occur, or to work in models of cryptography that still have some ideal, unrealizable aspect, in particular the random-oracle model. We discuss these possibilities and our choice in more detail in Section 3. The most important argument for our choice is that, depending on the timing assumptions possible in the environment and on the protocol goals, a range of different measures are conceivable for guaranteeing the necessary order between the sending of keys and ciphertexts. Further, existing formal methods and automated tools are well suited to arguing about such properties. Instead, if we proposed measures in the underlying idealization, we would need a once-and-for-all measure, and we would at present need to prove it by hand. To show the

[2] Given that one wants to achieve simulatability, the problem is independent of a surrounding framework for nested terms, i.e., of our specific goal of making the ideal encryption system a subsystem in the library of [4].

applicability of our choice for modeling the protocols typically analyzed in Dolev Yao models, we investigated the 50 protocols of the Clark-Jacob library [15] with respect to the commitment problem, and only one of them raises this problem.

Other design decisions to be taken with symmetric encryption are whether, given a ciphertext, the adversary may obtain information about the key used, and whether the ideal system prescribes that every decryption of a ciphertext (or a message of a different type) with the wrong key produces an error, or whether it may sometimes produce another message. Such questions are similar to the passive case in [2] or to the treatment of symmetric authentication in [5], but have to combined in a consistent way into the overall ideal encryption system.

2 Underlying Definitions

Before discussing the commitment problem in more detail, we present the exact definition of simulatability, the strong security notion that causes this problem. For this, we first briefly sketch the underlying definitions from [25]. This is the model used in the cryptographic library from [4] into which we embed our ideal encryption system.

A *system* consists of several possible *structures*. A structure consists of a set $\hat{M}$ of connected correct machines and a subset S of free ports, called *specified ports*. A machine is a probabilistic IO automaton (extended finite-state machine) in a slightly refined model to allow complexity considerations. For these machines Turing-machine realizations are defined, and the complexity of those is measured in terms of a common security parameter k, given as the initial work-tape content of every machine. Readers only interested in using the ideal cryptographic library (or even only the ideal encryption system) in larger protocols only need normal, deterministic IO automata.

In a *standard real cryptographic system*, the structures are derived from one intended structure and a trust model consisting of an access structure $\mathcal{ACC}$ and a channel model χ. Here $\mathcal{ACC}$ contains the possible sets $\mathcal{H}$ of indices of uncorrupted machines among the intended ones, and χ designates whether each channel is secure, authentic (but not private) or insecure. In a typical ideal system, each structure contains only one machine TH called *trusted host*.

Each structure is complemented to a *configuration* by an arbitrary *user* machine H and *adversary* machine A. H connects only to ports in S and A to the rest, and they may interact. The set of configurations of a system *Sys* is called $\mathsf{Conf}(Sys)$. The general scheduling model in [25] gives each connection c (from an output port $\mathsf{c}!$ to an input port $\mathsf{c}?$) a buffer, and the machine with the corresponding clock port $\mathsf{c}^{\triangleleft}!$ can schedule a message there when it makes a transition. In real asynchronous cryptographic systems, network connections are typically scheduled by A. A configuration is a runnable system, i.e., for each k one gets a well-defined probability space of *runs*. The *view* of a machine in a run is the restriction to all in- and outputs this machine sees and its internal states. Formally, the view $view_{conf}(\mathsf{M})$ of a machine M in a configuration *conf* is a *family of random variables* with one element for each security parameter value k.

2.1 Simulatability

Simulatability is the cryptographic notion of secure implementation. For reactive systems, it means that whatever might happen to an honest user in a real system Sys_{real} can also happen in the given ideal system Sys_{id}: For every structure $(\hat{M}_1, S) \in Sys_{\mathsf{real}}$, every polynomial-time user H, and every polynomial-time adversary A_1, there exists a polynomial-time adversary A_2 on a corresponding ideal structure $(\hat{M}_2, S) \in Sys_{\mathsf{id}}$ such that the view of H is computationally indistinguishable in the two configurations. This is illustrated in Figure 1. Indistinguishability is a well-known cryptographic notion from [27].

Definition 2.1 *(Computational Indistinguishability) Two families* $(\mathsf{var}_k)_{k\in\mathbb{N}}$ *and* $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ *of random variables on common domains* D_k *are* computationally indistinguishable *("$\approx$") iff for every algorithm* Dis *(the distinguisher) that is probabilistic polynomial-time in its first input,*

$$|P(\mathsf{Dis}(1^k, \mathsf{var}_k) = 1) - P(\mathsf{Dis}(1^k, \mathsf{var}'_k) = 1)| \in NEGL,$$

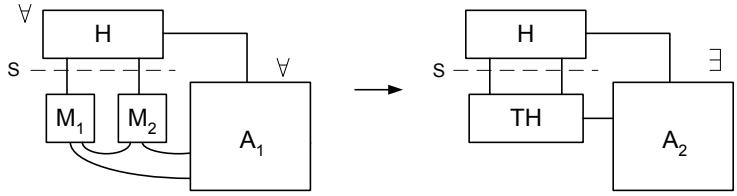

Figure 1: Simulatability: The two views of H must be indistinguishable.

where NEGL denotes the set of all negligible functions*, i.e.,* $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \in$ *NEGL iff for all positive polynomials* Q*,* $\exists k_0 \forall k \geq k_0\colon\ g(k) \leq 1/Q(k)$*.* ◇

Intuitively, given the security parameter and an element chosen according to either var_k or var'_k, Dis tries to guess which distribution the element came from.

Definition 2.2 *(Simulatability) For two systems* Sys_{real} *and* Sys_{id}*, one says* $Sys_{\mathsf{real}} \geq Sys_{\mathsf{id}}$ *(*at least as secure as*) iff for every polynomial-time configuration* $conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_{\mathsf{real}})$*, there exists a polynomial-time configuration* $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_{\mathsf{id}})$ *(with the same* H*) such that* $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$*.* ◇

For the cryptographic library, this is even shown with blackbox simulatability, i.e., A_2 consists of a simulator Sim that depends only on $(\hat{M}_1, S)$ and uses A_1 as a blackbox submachine. An essential feature of this definition of simulatability is a composition theorem [25], which roughly says that one can design and prove a larger system based on the ideal system Sys_{id}, and then securely replace Sys_{id} by the real system Sys_{real}.

2.2 Notation

We write ":=" for deterministic and "$\leftarrow$" for probabilistic assignment, and "$\xleftarrow{R}$" for uniform random choice from a set. By $x := y\texttt{++}$ for integer variables x, y we mean $y := y + 1; x := y$. The length of a message m is denoted as $\mathsf{len}(m)$, and $\downarrow$ is an error element available as an addition to the domains and ranges of all functions and algorithms. The list operation is denoted as $l := (x_1, \ldots, x_j)$, and the arguments are unambiguously retrievable as $l[i]$, with $l[i] = \downarrow$ if $i > j$. A database D is a set of functions, called entries, each over a finite domain called attributes. For an entry $x \in D$, the value at an attribute *att* is written $x.att$. For a predicate *pred* involving attributes, $D[pred]$ means the subset of entries whose attributes fulfill *pred*. If $D[pred]$ contains only one element, we use the same notation for this element. Adding an entry x to D is abbreviated $D :\Leftarrow x$.

2.3 Overview of the Ideal Cryptographic Library

The ideal cryptographic library as defined in [4] offers its users abstract cryptographic operations, such as commands to encrypt or decrypt a message, to make or test a signature, and to generate a nonce. All these commands have a simple, deterministic behavior in the ideal system. In a reactive scenario, this semantics is based on state, e.g., of who already knows which terms. State is stored in a "database". Each entry of the database has a type (e.g., "signature"), and pointers to its arguments (e.g., a key and a message). This corresponds to the top level of a Dolev-Yao term; an entire term can be found by following the pointers. Further, each entry contains handles for those participants who already know it. The reason for using handles to make an entry accessible for higher protocols is that an idealized cryptographic term and the corresponding real message have to be presented in the same way to higher protocols to allow for a provably secure implementation in the sense of simulatability. In the ideal library, handles essentially point to Dolev-Yao-like terms, while in the real library they point to cryptographic messages.

The ideal cryptographic library does not allow cheating by construction. For instance, if it receives a command to encrypt a message m with a certain key, it simply makes an abstract database entry for the ciphertext. Another user can only ask for decryption of this ciphertext if he has handles to both the ciphertext and the secret key. Similarly, if a user issues a command to sign a message, the ideal system looks

up whether this user should have the secret key. If yes, it stores that this message has been signed with this key. Later tests are simply look-ups in this database. A send operation makes an entry known to other participants, i.e., it adds handles to the entry. The underlying model does not only cover crypto operations, but it is an entire reactive system and therefore contains an abstract network model.

3 Design Decisions for Symmetric Encryption in Simulatability Proofs

In this section, we discuss several approaches to solve the commitment problem sketched in the introduction. We further elaborate on the main design decisions that we made to provide a suitable deterministic abstractions of symmetric encryption.

3.1 The Commitment Problem and Solution Approaches

As the name suggests, a "commitment problem" in simulatability proofs captures a situation where the simulator commits itself to a certain message and later has to change this commitment to allow for a correct simulation.

In the case of symmetric encryption, the commitment problem occurs if the simulator has to construct an indistinguishable ciphertext, knowing neither the secret key nor the plaintext used for the corresponding ciphertext in the real world. To simulate the missing key, the simulator will create a new secret key, or rely on an arbitrary, fixed key if the encryption systems guarantees indistinguishable keys, see [2]. Instead of the unknown plaintext, the simulator will encrypt an arbitrary message of the correct length, relying on the indistinguishability of ciphertexts of different messages. So far, the simulation is fine. It even stays fine if the message becomes known later because secure encryption still guarantees that it is indistinguishable that the simulator's ciphertext contains a wrong message. However, if the secret key becomes known later, the simulator runs into trouble, because, learning abstractly about this fact, it has to produce a suitable key that decrypts its ciphertext into the correct message. It cannot cheat with the message because it has to produce the correct behavior towards the honest users. This is typically not possible.

There is one perfect exception to the commitment problem, the one-time pad. For this specific encryption system, the simulator can open an arbitrary ciphertext string c to an arbitrary message m by selecting the key as $c \oplus m$. However, we do not want to restrict the ideal encryption system to modeling one-time pads, and for standard encryption systems and standard modes of operation, certainly no similar process is known. We can even show that no encryption system with fixed-length keys and a deterministic decryption algorithm can have this property. Assume there are x possible keys. Now let a protocol send fresh random messages whose overall length allows $2x$ possibilities. Hence if the simulator later has to produce a key, it has to be able to provide for $2x$ different cases of what the messages were with just x keys. Thus some of the keys it produces must fit more than one message tuple for the same given ciphertext tuple. "Fit" means in particular that the assumed deterministic decryption algorithm used by honest participants will in fact decrypt this ciphertext tuple to these messages. This is impossible.

The reason why the commitment problem did not occur in the cryptographic library before is that for public-key encryption it was enforced that secret keys are never sent, while for symmetric authentication it could be enforced that an authenticator never becomes known without the message it authenticates.

Related problems are known from uncoercible encryption and from adaptively secure multi-party function evaluation. However, none of the solutions provided there fits our case: Uncoercible encryption has even stronger requirements that need a physical assumption to be fulfilled at all [12]. Adaptively secure multi-party computation can either use deletion of old keys instead [8], or concentrates on public-key schemes [13, 7, 16], simply assuming the one-time pad for the symmetric case.

In the following, we introduce possible approaches to solve the commitment problem.

3.1.1 Assumptions about Sending Keys

Our aim is an abstraction that is as simple as possible and works for the cases typically analyzed in Dolev-Yao models. It turns out that for these typical cases, the commitment problem does not occur since the

overall protocol ensures that keys are not sent after having been used.

- Protocols with predistributed keys, as often assumed for authentication protocols, clearly fulfill this assumption. Formally, this can be seen as a synchronization assumption stating that the predistribution phase is over, at least per key, before one of the participants sharing this key starts using it.
- Synchronous protocols can make similar assumptions even if key exchange is part of the protocols, e.g., by indicating time bounds for the exchange phase and the usage phase for each key in the key exchange messages.
- Two-party protocols for exchanging a session key (using a symmetric or asymmetric master key) clearly fulfill the assumption if the party who generates the key sends it before using it. This is typically true.
- Three-party protocols where a key-distribution center S helps parties A and B to exchange a secret key sk come in several flavors: If S sends sk to A and B, and it can do so in one step (this depends on the detailed model of asynchrony), then the assumption is automatically fulfilled, and so it is if S sends sk to A, and A sends it to B and only then starts to use it. If S sends sk in two different steps, then the recipient of the first of these messages (or both if the first and second message look equal) has to wait for a confirmation from its partner before using the key. Many protocols have such a confirmation anyway.
- Many group key distribution protocols already have confirmation phases that can be used to fulfill the assumption.

To get a representative assessment of the restrictiveness of the commitment problem, we further investigated the protocols of the Clark-Jacob library [15]. From the 50 protocols of the library, only one—the (flawed) Wide Mouthed Frog protocol—raises the commitment problem. Further, avoiding the commitment problem is an integrity property that seems well within the scope of current automated protocol proof tools, so that it can be verified together with the application properties of a protocol. (This will become even clearer with the formal definition in Figure 2 and Definition 6.1.)

Hence our approach is to define a simple abstraction of symmetric encryption, and to show that it can be securely implemented provided that the commitment problem does not occur. The alternative approaches discussed below either exclude many more protocols, or require a much more complex abstraction, or rely on unrealistic assumptions like the random oracle model, or only work for special non-committing encryption schemes. An additional benefit of our solution is that we expect that our ideal encryption system also works with the random oracle model and non-committing encryption schemes, even for protocols that *do* have the commitment problem.

3.1.2 Internal Restrictions on Sending Keys

Another solution is to guarantee in the ideal and real system that the commitment problem cannot occur. This means that the systems only permit operations that do not cause the commitment problem, e.g., if a key has already been used for encrypting, it may no longer be sent. The problem is that this is a distributed property and thus not trivial to enforce in the real system. To implement it without imposing further restrictions on the patterns of how keys can be passed on, we might need Byzantine agreement before any participant first uses a key. This seems a highly unnatural underlying implementation for the authentication and key exchange protocols typically proved with Dolev-Yao models. It seems more natural to enforce restrictions on the patterns of how keys can be passed on. This certainly means that both the ideal system and the real system have to keep track of the current status of each key for each participant, e.g., whether it may still be sent. Furthermore, we either have to provide general rules for confirmation messages (compare Section 3.1.1) or to enforce patterns where no confirmation messages are needed. As we saw, the former already excludes some important cases, while the latter pulls distributed-systems aspects down into the cryptographic primitives. We therefore decided not to follow this approach.

3.1.3 Random Oracle Approach and Special Encryption Schemes

The commitment problem can be circumvented by conducting the proof in the random oracle model [10]. In a nutshell, including a random oracle in the encrypted message prevents the simulator from committing itself to a fixed value since the oracle can still be suitably instantiated when the commitment is opened respectively the secret key is sent. However, idealizations like random oracles do not capture cryptographic realities and protocols are known which are provably secure in these idealizations but insecure for any instantiation of the oracle [14], so that the benefit over simply using a Dolev-Yao model is not as great as we desire.

Further, the commitment problem does not occur for non-committing encryption schemes, but as we showed above, this currently only leaves the one-time pad.

3.2 The Need for Authenticated Encryption

Assume that a user encrypts a message m with one symmetric encryption key sk_1, and decrypts the resulting ciphertext with a key sk_2. In the ideal system, the result is the error symbol $\downarrow$ because no equation for this case is defined. In the real world, however, some encryption schemes yield another message m'. In particular, the one-time pad always yields a result. Similar problems are known from normal Dolev-Yao models, e.g., see [22].

We solve this problem by only considering encryption schemes that answer decryption requests with wrong keys with $\downarrow$, i.e., encryption schemes that provide a certain kind of authenticity. Formally, we use *authenticated symmetric encryption schemes* as defined in [11, 9]. They intuitively guarantee that if one does not know a specific key, it is infeasible to compute a ciphertext that can be validly decrypted with this key; see the definition in Section 5.1. This definition implies that decryption with a wrong key will always output $\downarrow$ except with negligible probability.

Instead of restricting the encryption schemes used, one could try to define the ideal system such that it allows non-error outputs for decryptions with wrong keys. However, a deterministic abstraction cannot achieve this because in the real system, decryption with different wrong keys will yield different messages if any, while in the ideal system all such wrong keys have a common abstraction. A non-deterministic choice could be achieved by letting the adversary make the choice of the resulting message, but this seems a somewhat undesirable ideal system, given that authenticated encryption is efficiently implementable under normal cryptographic assumptions.

3.3 Modeling Special Adversary Capabilities

Our idealization finally has to reflect special capabilities that the adversary may have with respect to symmetric encryption schemes in the real world.

First, we allow for checking whether encryptions have been created with the same secret key, as the definition of authenticated encryption schemes does not exclude that this can happen in the real system. For public-key encryption, this was achieved in [4] by tagging ciphertexts with the corresponding public key. For symmetric encryption, this is not possible as no public key exists. We solve this problem by tagging abstract ciphertexts with an otherwise meaningless "public key" solely used as an identifier for the secret key. An alternative approach was taken in [2] by only considering those encryption schemes that guarantee indistinguishable keys; for these schemes, this problem does not occur. However, if we wanted to restrict ourselves to this case we would first need to extend it to authenticated encryption.

Secondly, as encryption keys can also come from the adversary, it might happen that an encryption can be validly decrypted with several keys for incorrectly chosen keys. (The security definition only considers correct keys.) Hence it must be possible to tag encryptions with additional key identifiers during the execution of the ideal system. Encryptions without key identifiers model encryptions from the adversary for which no suitable key is known yet.

4 Ideal System

In the following, we present our ideal encryption system. We do this as an addition to the ideal cryptographic library reviewed in Section 2.3 for capturing symmetric encryption primitives. We stress that for modeling

and proving cryptographic protocols using our abstraction, it is sufficient to understand and use the ideal system described in this section. Later sections only justify the cryptographic faithfulness of this ideal library.

4.1 Structures and Parameters

The ideal system consists of a trusted host $\mathsf{TH}_{\mathcal{H}}$ for every subset $\mathcal{H}$ of a set $\{1, \ldots, n\}$ of users, denoting the possible honest users. It has a port $\mathsf{in}_u?$ for inputs from and a port $\mathsf{out}_u!$ for outputs to each user $u \in \mathcal{H}$ and for $u = \mathsf{a}$, denoting the adversary.

The ideal system keeps track of the length of messages using a tuple L of abstract length functions. We add functions $\mathsf{skse\_len}^*(k)$ and $\mathsf{symenc\_len}^*(k, l)$ to L for the length of symmetric encryption keys and ciphertexts, depending on a security parameter k and the length l of the message. Each function has to be polynomially bounded and efficiently computable.

4.2 States

The state of $\mathsf{TH}_{\mathcal{H}}$ consists of a database D and variables $size$, $curhnd_u$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$. The database D contains abstractions from real cryptographic objects which correspond to the top levels of Dolev-Yao terms. An entry has the following attributes:

- $x.ind \in \mathcal{INDS}$, called index, consecutively numbers all entries in D. We use the index as a primary key attribute of the database, i.e., we write $D[i]$ for the selection $D[ind = i]$.
- $x.type \in typeset$ identifies the type of x. We add types skse, pkse, and symenc to $typeset$ from [4], denoting secret symmetric encryption keys, corresponding "public keys", and symmetric encryptions. The type pkse is a so-called *secret type*, i.e., it must not be put into lists and hence cannot be transferred.
- $x.arg = (a_1, a_2, \ldots, a_j)$ is a possibly empty list of arguments. Many values a_i are indices of other entries in D and thus in $\mathcal{INDS}$. We sometimes distinguish them by a superscript "ind".
- $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ are handles by which a user or adversary u knows this entry. $x.hnd_u = \downarrow$ means that u does not know this entry. We use a superscript "hnd" for handles.
- $x.len \in \mathbb{N}_0$ denotes the "length" of the entry, which is computed by applying the functions from L.

Initially, D is empty. $\mathsf{TH}_{\mathcal{H}}$ has a counter $size \in \mathcal{INDS}$ for the current number of elements in D. New entries always receive $ind := size{+}{+}$, and $x.ind$ is never changed. For the handle attributes, it has counters $curhnd_u$ (current handle) initialized with 0, and each new handle for u will be chosen as $i^{\mathsf{hnd}} := curhnd{+}{+}$.

$\mathsf{TH}_{\mathcal{H}}$ further maintains explicit bounds on the length of messages and the number of activations to achieve polynomial runtime independent of the environment. The bounds from [4] can be used without modification except that the number of permitted inputs from the adversary has to be enlarged. This is just a technical detail to allow for a correct proof of simulatability. We omit further details.

4.3 New Inputs and their Evaluation

The ideal system has several types of inputs: *Basic commands* are accepted at all ports $\mathsf{in}_u?$; they correspond to cryptographic operations and have only local effects, i.e., only an output at the port $\mathsf{out}_u?$ for the same user occurs and only handles for u are involved. *Local adversary commands* are of the same type, but only accepted at $\mathsf{in}_{\mathsf{a}}?$; they model tolerated imperfections, i.e., possibilities that an adversary may have, but honest users do not. *Send commands* output values to other users. The notation $j \leftarrow \mathsf{algo}(i)$ for a command algo of $\mathsf{TH}_{\mathcal{H}}$ means that $\mathsf{TH}_{\mathcal{H}}$ receives an input $\mathsf{algo}(i)$ and outputs j if the input and output port are clear from the context. We only allow lists to be encrypted and transferred following a general convention in [4].

For symmetric encryption we add new basic commands and local adversary commands; the send commands are unchanged. We now define the precise new inputs and how $\mathsf{TH}_{\mathcal{H}}$ evaluates them based on its abstract state. Handle arguments are tacitly required to be in $\mathcal{HNDS}$ and existing, i.e., $\leq curhnd_u$, at the time of execution. The underlying model further bounds the length of each input to ensure polynomial

runtime; these bounds are not written out explicitly, but can easily be derived from the domain expectations given for the individual inputs.

The algorithm $i^{\mathsf{hnd}} \leftarrow \mathsf{ind2hnd}_u(i)$ (with side effect) denotes that $\mathsf{TH}_{\mathcal{H}}$ determines a handle i^{hnd} for user u to an entry $D[i]$: If $i^{\mathsf{hnd}} := D[i].hnd_u \neq \downarrow$, it returns that, else it sets and returns $i^{\mathsf{hnd}} := D[i].hnd_u := curhnd_u\mathsf{++}$. On non-handles, it is the identity function. $\mathsf{ind2hnd}_u^*$ applies $\mathsf{ind2hnd}_u$ to each element of a list.

4.3.1 Basic Commands

First we consider basic commands. This comprises operations for key generation, encryption, and decryption. We assume the current input is made at port $\mathsf{in}_u?$, and the result goes to $\mathsf{out}_u!$.

- *Key generation:* $skse^{\mathsf{hnd}} \leftarrow \mathsf{gen\_symenc\_key}()$. Set $skse^{\mathsf{hnd}} := curhnd_u\mathsf{++}$ and

$$\begin{aligned} D &:\Leftarrow (ind := size\mathsf{++}, type := \mathsf{pkse}, arg := (), len := 0); \\ D &:\Leftarrow (ind := size\mathsf{++}, type := \mathsf{skse}, arg := (ind-1), hnd_u := skse^{\mathsf{hnd}}, len := \mathsf{skse\_len}^*(k)). \end{aligned}$$

  The first entry, an "empty" public key without handle, serves as the mentioned key identifier for the secret key. The argument of the secret key "points" to the empty public key.

- *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{sym\_encrypt}(skse^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $skse := D[hnd_u = skse^{\mathsf{hnd}} \wedge type = \mathsf{skse}].ind$ and $l := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. Return $\downarrow$ if either of these is $\downarrow$, or if $length := \mathsf{symenc\_len}^*(k, D[l].len) > \mathsf{max\_len}(k)$. Otherwise, set $c^{\mathsf{hnd}} := curhnd_u\mathsf{++}$, $pkse := skse - 1$ and

$$D :\Leftarrow (ind := size\mathsf{++}, type := \mathsf{symenc}, arg := (l, pkse), hnd_u := c^{\mathsf{hnd}}, len := length).$$

  The general argument format for entries of type symenc is $((l_1, pkse_1), \ldots, (l_j, pkse_j))$. The arguments $pkse_1, \ldots, pkse_j$ are pairwise disjoint key identifiers of those secret keys for which the encryption validly decrypts into messages $l_1, \ldots, l_j$, respectively. We will see in Section 4.3.2 that additional key identifiers for an encryption can be added during the execution, e.g., since the adversary has created a suitable key. Such arguments are appended at the end of the existing list. An empty sequence of arguments models encryptions from the adversary for which no suitable secret key has been received yet.

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{sym\_decrypt}(skse^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  If $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{symenc}].ind = \downarrow$ or $skse := D[hnd_u = skse^{\mathsf{hnd}} \wedge type = \mathsf{skse}].ind = \downarrow$, return $\downarrow$. Otherwise, let $((l_1, pkse_1), \ldots, (l_j, pkse_j)) := D[c].arg$ (where j may be 0). If $skse - 1 = pkse_i$ for some $1 \leq i \leq j$, set $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l_i)$ else $l^{\mathsf{hnd}} := \downarrow$.

4.3.2 Local Adversary Commands

The following local commands are only accepted at the port $\mathsf{in}_\mathsf{a}?$. They model special capabilities of the adversary, see Section 3.3. For dealing with symmetric encryptions from the adversary for which no suitable key has been received yet, we provide a command for generating an *unknown symmetric encryption*. Later, suitable secret keys may be received. A command for *fixing symmetric encryptions* takes care of this. Finally, we allow the adversary to retrieve all information that we do not explicitly require to be hidden, e.g., arguments and the type of a given handle. For this, we extend the general command for *parameter retrieval* for the symmetric encryption system. For entries of type symenc, only the length of the encrypted message is output instead of the message itself unless the adversary has the corresponding secret-key handle.

- *Unknown symmetric encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{adv\_unknown\_symenc}(length)$ with $length \in \mathbb{N}$.

  Return $\downarrow$ if $length > \mathsf{max\_len}(k)$. Set $c^{\mathsf{hnd}} := curhnd_a\mathsf{++}$ and

$$D :\Leftarrow (ind := size\mathsf{++}, type := \mathsf{symenc}, arg := (), hnd_\mathsf{a} := c^{\mathsf{hnd}}, len := length).$$

- *Fixing symmetric encryption:* $v \leftarrow \mathsf{adv\_fix\_symenc\_content}(skse^{\mathsf{hnd}}, c^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Return $\downarrow$ if $c := D[hnd_{\mathsf{a}} = c^{\mathsf{hnd}} \wedge type = \mathsf{symenc}].ind = \downarrow$, if $skse := D[hnd_u = skse^{\mathsf{hnd}} \wedge type = \mathsf{skse}].ind = \downarrow$, if $l := D[hnd_{\mathsf{a}} = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind = \downarrow$, or if $\mathsf{symenc\_len}^*(k, D[l].len) \neq D[c].len$.

  Let $pkse := skse - 1$ and $((l_1, pkse_1), \ldots, (l_j, pkse_j)) := D[c].arg$ (where j may be 0). If $pkse \notin \{pkse_1, \ldots, pkse_j\}$ then set $D[c].arg := ((l_1, pkse_1), \ldots, (l_j, pkse_j), (l, pkse))$ and $v := \mathsf{true}$, else set $v := \mathsf{false}$.

- *Parameter retrieval:* $(type, arg) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$.

  This existing command always sets $type := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].type$, and for most types $arg := \mathsf{ind2hnd}^*_{\mathsf{a}}(D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].arg)$. This also applies to the new types skse and pkse. For $type = \mathsf{symenc}$, let $((l_1, pkse_1), \ldots, (l_j, pkse_j)) := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].arg$. For $i \in \{1, \ldots, j\}$, let $pkse_i^{\mathsf{hnd}} := \mathsf{ind2hnd}_{\mathsf{a}}(pkse_i)$ and $skse_i := pkse_i + 1$. Then if $D[skse_i].hnd_{\mathsf{a}} \neq \downarrow$, let $l'_i := \mathsf{ind2hnd}_{\mathsf{a}}(l_i)$, else $l'_i := D[l_i].len$. Finally let $arg := ((l'_1, pkse_1^{\mathsf{hnd}}), \ldots, (l'_j, pkse_j^{\mathsf{hnd}}))$.

For unknown encryptions, neither a key identifier nor a message exists. Note further that parsing a symmetric encryption yields handles to the "empty" public keys. For an encryption generated by an honest user, the first public key always corresponds to the secret key with which the encryption was generated. If the adversary wants to know whether two encryptions were created using the same secret key, it parses them and compares the resulting public keys.

4.3.3 Send Commands

The ideal cryptographic library offers commands for virtually sending messages to other users. Sending is modeled by adding a new handle for the intended recipient and possibly one for the adversary to the database entry modeling the message. These handles always point to a list entry, which can contain arbitrary application data, ciphertexts, public keys, etc., and now also symmetric encryptions and the corresponding secret keys. These commands are unchanged from [4]; as an example we present those modeling insecure channels, which are the most commonly used ones, and omit secure channels and authentic channels.

- $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$. Intuitively, the list l shall be sent to user v. Let $l^{\mathsf{ind}} := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l^{\mathsf{ind}} \neq \downarrow$, then output $(u, v, \mathsf{ind2hnd}_{\mathsf{a}}(l^{\mathsf{ind}}))$ at $\mathsf{out}_{\mathsf{a}}$!.

- $\mathsf{adv\_send\_i}(u, v, l^{\mathsf{hnd}})$, for $u \in \{1, \ldots, n\}$ and $v \in \mathcal{H}$ at port in_{a}?. Intuitively, the adversary wants to send list l to v, pretending to be u. Let $l^{\mathsf{ind}} := D[hnd_{\mathsf{a}} = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l^{\mathsf{ind}} \neq \downarrow$ output $(u, v, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at out_v!.

5 Real System

The real cryptographic library offers its users the same commands as the ideal one, i.e., honest users operate on cryptographic objects via handles. There is one separate machine with a database for each honest user in the real system, containing real cryptographic keys, real encryptions, etc.. Real bitstrings are actually sent between machines. The commands are implemented by real cryptographic algorithms, and the simulatability proof will show that nevertheless, everything a real adversary can achieve can also be achieved by an adversary in the ideal system, or otherwise the underlying cryptography can be broken. We now present our additions and modifications to the real system of [4], starting with a description of the underlying cryptographic definitions.

5.1 Underlying Cryptographic Operations

We denote a symmetric encryption scheme by a tuple $\mathcal{SE} = (\mathsf{gen_{SE}}, \mathsf{sym\_encrypt}, \mathsf{sym\_decrypt}, \mathsf{skse\_len}, \mathsf{symenc\_len})$ of polynomial-time algorithms. Key generation for a security parameter $k \in \mathbb{N}$ is written as

$$sk \leftarrow \mathsf{gen_{SE}}(1^k).$$

The length of sk is $\mathsf{skse\_len}(k) > 0$. We denote the encryption of a message $m \in \{0,1\}^+$ by

$$c \leftarrow \mathsf{sym\_encrypt}_{sk}(m)$$

and decryption by

$$m := \mathsf{sym\_decrypt}_{sk}(c).$$

The result may be $\downarrow$; then we call the ciphertext invalid for this key. A correctly generated ciphertext for a key of the correct length always has to be valid for this key.

The length of c is $\mathsf{symenc\_len}(k, \mathsf{len}(m)) > 0$. This is also true for every c' with $\mathsf{sym\_decrypt}_{sk}(c') \neq \downarrow$ for a value $sk \in \{0,1\}^{\mathsf{skse\_len}(k)}$. The functions $\mathsf{skse\_len}$ and $\mathsf{symenc\_len}$ must be bounded by multivariate polynomials. Our requirement that such functions exist is without loss of generality due to standard padding techniques.

Our security definition is the standard definition for authenticated symmetric encryption schemes from [11, 9]. It consists of two parts: The scheme must ensure confidentiality of messages under chosen-ciphertext attacks, and it must guarantee integrity of ciphertexts. In the following, we formulate these notions using the notation for interacting machines.

Definition 5.1 *(Security against Chosen-Ciphertext Attacks) Given a symmetric encryption scheme, the* symmetric decryptor machine SymDec *is defined as follows: It has one input and one output port, a variable sk, initialized with $\downarrow$, an initially empty set C and the following transition rules:*

- *First generate a key as $sk \leftarrow \mathsf{gen}_{\mathsf{SE}}(1^k)$ and set $b \xleftarrow{R} \{0,1\}$.*
- *On input $(\mathsf{symenc}, m_0, m_1)$ (intuitively a pair of messages an adversary hopes to be able to distinguish), and if $\mathsf{len}(m_0) = \mathsf{len}(m_1)$, set $c \leftarrow \mathsf{sym\_encrypt}_{sk}(m_b)$, $C := C \cup \{c\}$, and output c.*
- *On input (symdec, c_j) and if $c_j \notin C$, return $\mathsf{sym\_decrypt}_{sk}(c_j)$.*

The encryption scheme is called indistinguishable under chosen-ciphertext attack *if for every probabilistic polynomial-time machine $\mathsf{A_{SD}}$ that interacts with* SymDec *and finally outputs a bit b^* (meant as a guess at b), the probability of the event $b^* = b$ is bounded by $1/2 + g(k)$ for a negligible function g.* ◇

The machine for defining integrity of ciphertexts is defined similarly.

Definition 5.2 *(Integrity of Ciphertexts) Given a symmetric encryption scheme, we define the* symmetric integrity machine SymInt *as follows: It has one input and one output port, a variable sk initialized with $\downarrow$, and the following transition rules:*

- *First generate a key as $sk \leftarrow \mathsf{gen}_{\mathsf{SE}}(1^k)$.*
- *On input (symenc, m_j), return $c_j \leftarrow \mathsf{sym\_encrypt}_{sk}(m_j)$.*
- *On input (symdec, c'_j), return $m'_j := \mathsf{sym\_decrypt}_{sk}(c'_j)$.*

The encryption scheme is said to have integrity of ciphertexts *if for every probabilistic polynomial-time machine $\mathsf{A_{SI}}$ that interacts with* SymInt *the probability is negligible (in k) that* SymInt *outputs $m \neq \downarrow$ on any input (symdec, c) where c was not output by* SymInt *upon a command $(\mathsf{symenc}, \cdot)$ until that time, i.e., not among the c_j's.* ◇

Symmetric encryptions schemes provably secure with respect to these two definitions exist under reasonable assumptions [26]. Bellare and Namprempre even showed in [9] that such encryption schemes can be derived from any symmetric encryption scheme that is provably secure under adaptive chosen-*plaintext* attacks together with any strongly unforgeable message authentication code by first encrypting a plaintext and then appending a MAC to the obtained ciphertext.

5.2 Structures

The intended structure of the real cryptographic library consists of n machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$. Each M_u has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$, so that the same honest users can connect to the ideal and the real system. Each M_u has connections to each M_v exactly as in [4], in particular an insecure connection called $\mathsf{net}_{u,v,\mathsf{i}}$ for normal use. They are called network connections and the corresponding ports network ports. Any subset $\mathcal{H}$ of $\{1, \ldots, n\}$ can denote the indices of correct machines. The resulting actual structure consists of the correct machines with modified channels according to a channel model. In particular, an insecure channel is split in the actual structure so that both machines actually interact with the adversary. Details of the channel model are not needed here. Such a structure then interacts with honest users H and an adversary A.

5.3 States of a Machine

The state of each machine M_u consists of a database D_u and a variable $curhnd_u$. Each entry x in D_u has the following attributes:

- $x.hnd_u \in \mathcal{HNDS}$ consecutively numbers all entries in D_u. We use it as a primary key attribute, i.e., we write $D_u[i^{\mathsf{hnd}}]$ for the selection $D_u[hnd_u = i^{\mathsf{hnd}}]$.
- $x.word \in \{0,1\}^+$ is the real representation of x.
- $x.type \in typeset \cup \{\mathsf{null}\}$ identifies the type of x, where the value null denotes an unparsed entry.
- $x.add\_arg$ is a list of ("additional") arguments. For entries of our new types it is always ().

Initially, D_u is empty. M_u has a counter $curhnd_u \in \mathcal{HNDS}$ for the current size of D_u. The subroutine

$$(i^{\mathsf{hnd}}, D_u) :\leftarrow (i, type, add\_arg)$$

determines a handle for certain given parameters in D_u: If an entry with the word i already exists, i.e., $i^{\mathsf{hnd}} := D_u[word = i \wedge type \notin \{\mathsf{sks}, \mathsf{ske}\}].hnd_u \neq \downarrow$,[3] it returns i^{hnd}, assigning the input values $type$ and add_arg to the corresponding attributes of $D_u[i^{\mathsf{hnd}}]$ only if $D_u[i^{\mathsf{hnd}}].type$ was null. Else if $\mathsf{len}(i) > \mathsf{max\_len}(k)$, it returns $i^{\mathsf{hnd}} = \downarrow$. Otherwise, it sets and returns $i^{\mathsf{hnd}} := curhnd_u\mathsf{++}$, $D_u :\Leftarrow (i^{\mathsf{hnd}}, i, type, add\_arg)$.

Similar to the machine $\mathsf{TH}_{\mathcal{H}}$, M_u maintains explicit bounds on the length of messages and number of activations to achieve polynomial runtime. We omit further details.

5.4 Inputs and their Evaluation

Now we describe how M_u evaluates individual new inputs.

5.4.1 Constructors and One-level Parsing

The stateful commands are defined via functional constructors and parsing algorithms for each type. A general functional algorithm

$$(type, arg) \leftarrow \mathsf{parse}(m),$$

then parses arbitrary entries as follows: It first tests if m is of the form $(type, m_1, \ldots, m_j)$ with $type \in typeset \setminus \{\mathsf{pkse}, \mathsf{pka}, \mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ and $j \geq 0$. If not, it returns $(\mathsf{garbage}, ())$. Otherwise it calls a type-specific parsing algorithm $arg \leftarrow \mathsf{parse\_}type(m)$. If the result is $\downarrow$, parse again outputs $(\mathsf{garbage}, ())$. By

$$\text{"parse } m^{\mathsf{hnd}}\text{"}$$

we abbreviate that M_u calls $(type, arg) \leftarrow \mathsf{parse}(D_u[m^{\mathsf{hnd}}].word)$, assigns $D_u[m^{\mathsf{hnd}}].type := type$ if it was still null, and may then use arg. By

$$\text{"parse } m^{\mathsf{hnd}} \text{ if necessary"}$$

we mean the same except that M_u does nothing if $D_u[m^{\mathsf{hnd}}].type \neq \mathsf{null}$.

[3] The restriction $type \notin \{\mathsf{sks}, \mathsf{ske}\}$ (abbreviating secret keys of signature and public-key encryption schemes) is included for compatibility to the original library. Similar statements will occur some more times, but no further knowledge of such types is needed for understanding the new work.

5.4.2 Basic Commands and parse_*type*

First we consider basic commands. They are again local. In M_u this means that they produce no outputs at the network ports. The term "tagged list" means a valid list of the real system. We assume that tagged lists are efficiently encoded into $\{0,1\}^+$.

- *Key constructor:* $sk^* \leftarrow \mathsf{make\_symenc\_key}()$.
  Let $sk \leftarrow \mathsf{gen}_{\mathsf{SE}}(1^k)$, $sr \xleftarrow{R} \{0,1\}^{\mathsf{nonce\_len}(k)}$, and return $sk^* := (\mathsf{skse}, sk, sr)$.
- *Key generation:* $skse^{\mathsf{hnd}} \leftarrow \mathsf{gen\_symenc\_key}()$.
  Let $sk^* \leftarrow \mathsf{make\_symenc\_key}()$, $skse^{\mathsf{hnd}} := curhnd_u\mathsf{++}$, and $D_u :\Leftarrow (skse^{\mathsf{hnd}}, sk^*, \mathsf{skse}, ())$.
- *Key parsing:* $arg \leftarrow \mathsf{parse\_skse}(sk^*)$.
  If sk^* is of the form (skse, sk, sr) with $sk \in \{0,1\}^{\mathsf{skse\_len}(k)}$ and $sr \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, return $()$, else $\downarrow$.
- *Symmetric encryption constructor:* $c^* \leftarrow \mathsf{make\_symenc}(sk^*, l)$, for $sk^*, l \in \{0,1\}^+$.
  Set $r \xleftarrow{R} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $sk := sk^*[2]$, and $sr := sk^*[3]$. Encrypt as $c \leftarrow \mathsf{sym\_encrypt}_{sk}((r,l))$, and return $c^* := (\mathsf{symenc}, sr, r, c)$.
- *Symmetric encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{sym\_encrypt}(skse^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.
  Parse $skse^{\mathsf{hnd}}$ and l^{hnd} if necessary. If $D_u[skse^{\mathsf{hnd}}].type \neq \mathsf{skse}$ or $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, then return $\downarrow$. Otherwise set $sk^* := D_u[skse^{\mathsf{hnd}}].word$, $l := D_u[l^{\mathsf{hnd}}].word$, and $c^* \leftarrow \mathsf{make\_symenc}(sk^*, l)$. If $\mathsf{len}(c^*) > \mathsf{max\_len}(k)$, return $\downarrow$, else set $c^{\mathsf{hnd}} := curhnd_u\mathsf{++}$ and $D_u :\Leftarrow (symenc^{\mathsf{hnd}}, c^*, \mathsf{symenc}, ())$.
- *Encryption parsing:* $arg \leftarrow \mathsf{parse\_symenc}(c^*)$.
  If c^* is not of the form $(\mathsf{symenc}, sr, r, c)$ with $sr, r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$ and $c \in \{0,1\}^+$, return $\downarrow$, else set $arg := ()$.
- *Symmetric decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{sym\_decrypt}(c^{\mathsf{hnd}}, skse^{\mathsf{hnd}})$.
  Parse c^{hnd} and $skse^{\mathsf{hnd}}$. If $D_u[c^{\mathsf{hnd}}].type \neq \mathsf{symenc}$ or $D_u[skse^{\mathsf{hnd}}].type \neq \mathsf{skse}$, return $\downarrow$. Else let $(\mathsf{symenc}, sr, r, c) := D_u[c^{\mathsf{hnd}}].word$ and $sk := D_u[skse^{\mathsf{hnd}}].word[2]$. Let $l^* := \mathsf{sym\_decrypt}_{sk}(c)$ and $l := l^*[2]$. If $sr \neq D_u[skse^{\mathsf{hnd}}].word[3]$, or $l^* = \downarrow$, or $l^*[1] \neq r$, or if l is not a tagged list, return $l^{\mathsf{hnd}} := \downarrow$. Otherwise let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

5.4.3 Send Commands and Network Inputs

Similar to the ideal system, there is a command $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$ for sending a list l from u to v, but now using the port $\mathsf{net}_{u,v,\mathsf{i}}!$, i.e., using the real insecure network: On input $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$ for $v \in \{1, \ldots, n\}$, M_u parses l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$, M_u outputs $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,\mathsf{i}}!$.

Inputs at network ports are simply tested for being tagged lists and stored as in [4].

6 Security Proof

Our security claim is that the real cryptographic library extended with symmetric encryption is as secure as the ideal cryptographic library with symmetric encryption in the sense of Definition 2.2 provided that the commitment problem is avoided by the surrounding protocol.

We first have to define what it means that the commitment problem does not occur. We formalize the following event NoComm: if there exists an input at a specified port that causes a symmetric encryption to be generated such that the corresponding key is not known to the adversary, then future inputs may only cause this key to be sent within an encryption that cannot be decrypted by the adversary. Note that this property could still be marginally weakened by restricting it to those cases where the symmetric encryption is actually sent to the adversary; however, our variant is easier to verify for actual protocols since one does not have to additionally parse every sent term to look for a contained encryption. For technical reasons, we further exclude encryption cycles (such as encrypting a key with itself) within the definition of NoComm,

If there exists $t_1 \in \mathbb{N}$, $i \in \mathcal{INDS}$, $u_1 \in \mathcal{H}$ such that for $skse^{\mathsf{hnd}}_{u_1} := D[i].hnd_{u_1}$, we have

$t_1 : \mathsf{in}_{u_1}?.\mathsf{sym\_encrypt}(skse^{\mathsf{hnd}}_{u_1}, l^{\mathsf{hnd}}_1)$ and — # *If a term is encrypted at time* t_1
$t_1 : D[i].type = \mathsf{skse}$ and — # *with a secret key*
$t_1 : D[i].hnd_{\mathsf{a}} = \downarrow$ — # *that is not known to the adversary*

then the following must hold. For every $t_2 > t_1$, v_2, $u_2 \in \mathcal{H}$ we have

$t_2 : \mathsf{in}_{u_2}?.\mathsf{send\_A}(l^{\mathsf{hnd}}_2, v_2)$ — # *If another term is sent at time* t_2 *and*
$D[i] \in \mathsf{tree}(t_2 : D[hnd_{u_2} = l^{\mathsf{hnd}}_2])$ — # *and the secret key is contained in this term*
$\Longrightarrow$ — # *then*
$t_2 : \mathsf{wrapped}(i, t_2 : D[hnd_{u_2} = l^{\mathsf{hnd}}_2].ind)$. — # *the secret key is sufficiently wrapped*

Figure 2: The property NoComm.

which had to be required even for acquiring properties weaker than simulatability. We refer to [2] for further discussions.

To capture the event NoComm formally, we first define the tree of contained terms of a database entry $D[i]$, written $\mathsf{tree}(D[i])$, by defining that $D[i]$ is the root of the tree, and $D[j]$ is a child of $D[k]$ if and only if $j \in D[k].arg$. We recall that symmetric encryptions do not maintain the secret keys used for generating these encryptions as arguments but only the corresponding public-key identifiers. To capture the absence of encryption cycles, we define a function order on honestly generated secret encryption keys that are not known to the adversary when they are first used. The function order then assigns each key a number corresponding to the order in which the keys are first used for encryption. We also define that honestly generated secret keys of public-key encryption schemes are always of order 0. Later on, we will require that a key of order i may only be encrypted by keys of order $j < i$.

The event NoComm is formally defined in Figure 2. Here, a statement of the form "$t : \mathsf{p}?.\mathsf{send\_A}(l^{\mathsf{hnd}}, v)$" means that a send command is input at port p? of $\mathsf{TH}_{\mathcal{H}}$ at time t so that the sent term will be received by the adversary. Formally, this means that v can be arbitrary for sending on insecure or authentic channels, and that v has to be dishonest for sending on secure channels. We further write $t : D$ to describe the contents of database D at time t. A statement of the form $t : \mathsf{wrapped}(j, i)$ is true if and only if for every occurrence of the node $D[j]$ in $\mathsf{tree}(t : D[i])$ with $t : D[j].type = \mathsf{skse}$ there exists a node $D[k]$ in $\mathsf{tree}(t : D[i])$ such that $t : D[k].type \in \{\mathsf{symenc}, \mathsf{enc}\}$, $D[j]$ is a descendant of $D[t : D[k].arg[1]]$ (i.e., of the encrypted message), $D[k].hnd_{\mathsf{a}} = \downarrow$ and $t : \mathsf{order}(sk) < t : \mathsf{order}(j)$ where sk denotes the secret key used for encrypting the message, i.e., $sk := t : D[k].arg[1][2] + 1$ if $t : D[k].type = \mathsf{symenc}$ respectively $sk := t : D[k].arg[2] - 1$ if $t : D[k].type = \mathsf{enc}$.

Is is easy to see that one could as well define the event NoComm only in terms of the inputs that $\mathsf{TH}_{\mathcal{H}}$ obtains from the honest users, i.e., independent of the state of $\mathsf{TH}_{\mathcal{H}}$ and solely depending on the interaction with the surrounding protocol. However, this description would be very lengthy and is hence omitted for reasons of readability.

We now define those configurations to be *commitment-free* in which the event NoComm holds independent of the considered adversary, i.e., where the honest user already guarantees the validity of the event. As the event can be restated in terms of the inputs obtained from the user, commitment-free configuration are naturally also defined for the real library as it offers the same ports and commands to the honest users as the ideal library.

Definition 6.1 (Commitment-free Configurations and Simulatability) *A user* H *is* commitment-free *with respect to symmetric encryption and the machine* $\mathsf{TH}_{\mathcal{H}}$ *if for all configurations* $conf = (\mathsf{TH}_{\mathcal{H}}, S_{\mathcal{H}}, \mathsf{H}, \mathsf{A})$*, the property* NoComm *as defined in Figure 2 holds. Configurations with a commitment-free user are called* commitment-free configurations*. The restriction of simulatability to the set of commitment-free configurations is denoted by* $\geq^{\mathsf{Comm}}$*, i.e., for all commitment-free configurations of the real system, there exists a commitment-free configuration of the ideal system with the same honest user that achieves indistinguishable*

views for the honest user. ◇

Let $RPar$ be the set of valid parameter tuples for the real system, consisting of the number $n \in \mathbb{N}$ of participants, secure signature, encryption, and symmetric encryption schemes $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{SE}$, and length functions and bounds L'. For $(n, \mathcal{S}, \mathcal{E}, \mathcal{SE}, L') \in RPar$, let $Sys^{\mathsf{cry\_sym,real}}_{n,\mathcal{S},\mathcal{E},\mathcal{SE},L'}$ be the resulting real cryptographic library. Further, let the corresponding length functions and bounds of the ideal system be formalized by a function $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, \mathcal{SE}, L')$, and let $Sys^{\mathsf{cry\_sym,id}}_{n,L}$ be the ideal cryptographic library with parameters n and L. The extension of R2Ipar to the newly added length functions for symmetric encryption, i.e., skse_len* and symenc_len* is given in Appendix B. Using the notation of Definition 2.2 and 6.1, we have

Theorem 6.1 *(Security of Cryptographic Library) For all parameters $(n, \mathcal{S}, \mathcal{E}, \mathcal{SE}, L') \in RPar$, we have*

$$Sys^{\mathsf{cry\_sym,real}}_{n,\mathcal{S},\mathcal{E},\mathcal{SE},L'} \geq^{\mathsf{Comm}} Sys^{\mathsf{cry\_sym,id}}_{n,L},$$

where $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, \mathcal{SE}, L')$. □

For proving this theorem for the original library without symmetric encryption, a simulator $\mathsf{Sim}_{\mathcal{H}}$ has been defined in [4] such that even the combination of arbitrary polynomial-time users H and an arbitrary polynomial-time adversary A cannot distinguish the combination of the real machines M_u from the combination $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ (for all sets $\mathcal{H}$ indicating the correct machines). We sketch how we extend the simulator and then the proof of correct simulation to deal with symmetric encryption. A fully rigorous definition of $\mathsf{Sim}_{\mathcal{H}}$ is postponed to Appendix A.

6.1 Simulator

Basically $\mathsf{Sim}_{\mathcal{H}}$ has to translate real messages from the real adversary A into handles as $\mathsf{TH}_{\mathcal{H}}$ expects them at its adversary input port in_{a}? and vice versa. In both directions, $\mathsf{Sim}_{\mathcal{H}}$ has to parse an incoming message completely because it can only construct the other version (abstract or real) bottom-up. This is done by recursive algorithms. The state of $\mathsf{Sim}_{\mathcal{H}}$ mainly consists of a database D_{a}, similar to the databases D_u, but storing the knowledge of the adversary. The behavior of $\mathsf{Sim}_{\mathcal{H}}$ is sketched as follows.

Inputs from $\mathsf{TH}_{\mathcal{H}}$. Assume that $\mathsf{Sim}_{\mathcal{H}}$ receives an input $(u, v, x, l^{\mathsf{hnd}})$ from $\mathsf{TH}_{\mathcal{H}}$. If a bitstring l for l^{hnd} already exists in D_{a}, i.e., this message is already known to the adversary, the simulator immediately outputs l at port $\mathsf{net}_{u,v,x}$!. Otherwise, it first constructs such a bitstring l with a recursive algorithm id2real. This algorithm decomposes the abstract term using basic commands and the adversary command adv_parse. At the same time, id2real builds up a corresponding real bitstring using real cryptographic operations and enters all new message parts into D_{a} to recognize them when they are reused, both by $\mathsf{TH}_{\mathcal{H}}$ and by A.

We sketch how the simulator is extended to deal with symmetric encryption keys respectively symmetric encryptions. If the entry corresponding to l^{hnd} is a symmetric encryption key, id2real creates a new secret key by applying the function make_symenc_key and uses this key whenever an abstract encryption has to be simulated under the abstract key entry l^{hnd}. If the entry corresponding to l^{hnd} is a symmetric encryption, $\mathsf{Sim}_{\mathcal{H}}$ first determines the corresponding secret key by means of the public key identifier of the encryption. After that, it checks whether the designated recipient of the handle is a dishonest or an honest party. In the first case, adv_parse reveals the plaintext of the encrypted message, so id2real only has to encrypt this plaintext with the determined secret key and output this encryption. If the designated recipient is honest, then adv_parse only outputs the length of the encrypted message. In this case, id2real encrypts a fixed message of equal length.

Inputs from A. Now assume that $\mathsf{Sim}_{\mathcal{H}}$ receives a bitstring l from A at a port $\mathsf{net}_{u,v,x}$?. If l is not a valid list, $\mathsf{Sim}_{\mathcal{H}}$ aborts the transition. Otherwise it translates l into a corresponding handle l^{hnd} by an algorithm real2id, and outputs the abstract sending command $\mathsf{adv\_send}\_x(w, u, l^{\mathsf{hnd}})$ at port in_{a}!.

If a handle l^{hnd} for l already exists in D_{a}, then real2id reuses that. Otherwise it recursively parses a real bitstring using the functional parsing algorithm. At the same time, it builds up a corresponding abstract term in the database of $\mathsf{TH}_{\mathcal{H}}$. This finally yields the handle l^{hnd}. Furthermore, real2id enters all new subterms into

D_{a}. For building up the abstract term, real2id makes extensive use of the special capabilities of the adversary modeled in $\mathsf{TH}_{\mathcal{H}}$. In the real system, the bitstring may, e.g., contain an encryption which no encryption key is known yet that could valid decrypt this encryption. Therefore, the simulator has to be able to insert such an encryption with unknown key and unknown plaintext into the database of $\mathsf{TH}_{\mathcal{H}}$, which explains the need for the command adv_unknown_symenc. Similarly, the adversary might send a new encryption key which has to be added to existing symmetric encryption entries for which this key is valid. All these and similar cases for symmetric encryption can be covered by using the special adversary capabilities that we offered in Section 4.3.2.

6.2 Proof of Correct Simulation

In the proof of the extended cryptographic library, now including symmetric encryption, we retain the original proof structure as far as possible. The basic structure of that proof is that a combined system $\mathsf{C}_{\mathcal{H}}$ is defined that essentially contains all aspects of both the real and the ideal system, and then bisimulations are proved between $\mathsf{C}_{\mathcal{H}}$ and the combination $\mathsf{M}_{\mathcal{H}}$ of the real machines, and between $\mathsf{C}_{\mathcal{H}}$ and the combination $\mathsf{THSim}_{\mathcal{H}}$ of the trusted host and the simulator. A bisimulation, however, cannot deal with computational indistinguishability. Hence at the beginning of the proof, the real asymmetric encryptions were replaced by simulated ones as made in the simulator. This could be done in one replacement step, using a low-level idealization of asymmetric encryption and the composition theorem. The overall proof is illustrated in Figure 3 where Steps 1 and 2 depict the treatment of public-key encryption, and where Step 4 and the system $\mathsf{C}^*_{\mathcal{H}}$ were not present in the original proof.

Symmetric encryption is more complicated because we also allow symmetric keys to be sent around. However, a typical low-level idealization would assume, like the original cryptographic definitions of encryption security, that the keys are only used for correct en- and decryption. Intuitively, this is OK in our case because the simulator treats keys that the adversary learns perfectly correctly, and if the adversary does not learn a key, i.e., the key is never sent at all or only encrypted, then it should be as good as if it had never been used apart from en- and decryption. However, here we argue with the security of encryption while trying to show the security of encryption, and we must ensure that the argument is not circular. Fortunately, our assumptions guarantee that we always argue with the security of encryption with another key when treating one key, and that the keys can be arranged in non-circular order for this treatment.

We therefore perform a successive exchange of real encryptions for simulated encryptions by a so-called hybrid argument. We do this in the combined system because there we have all information easily available, in particular, which keys are ideally known to the adversary. In the overall proof depicted in Figure 3, Step 4 and the fact that there are multiple indexed combined systems $\mathsf{C}^{(i)}_{\mathcal{H}}$ are the new aspects for symmetric encryption.

6.2.1 Initial and Final Combined Systems

The initial combined system $\mathsf{C}_{\mathcal{H}}$ is defined indirectly from the real and ideal system exactly as in [4]. In particular, it contains a database D^* that extends the database D of TH by an attribute *word* containing real word entries as in $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{Sim}_{\mathcal{H}}$. These real words are computed as in $\mathsf{M}_{\mathcal{H}}$ for entries generated by basic commands, i.e., by the honest users, while they are computed as in $\mathsf{Sim}_{\mathcal{H}}$ for entries resulting from network inputs, i.e., values coming from the adversary. This implies that all symmetric encryptions produced by honest users contain a real plaintext message.

The final combined system $\mathsf{C}^*_{\mathcal{H}}$ is equal to $\mathsf{C}_{\mathcal{H}}$ except for symmetric encryptions: For encryptions made by honest users and with keys of honest users, a simulated message 1^{len^*} defined as in $\mathsf{Sim}_{\mathcal{H}}$ is encrypted instead of a real plaintext message. To distinguish keys generated by honest users from keys generated by the adversary within both $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{C}^*_{\mathcal{H}}$, we give entries of type skse an additional attribute *owner* ranging over {honest, adv}, which captures if this key has been generated by an honest user or by the adversary. This means that if a command gen_symenc_key is input at $\mathsf{in}_u?$, then in the new entry x both systems additionally set $x.owner :=$ honest for $u \in \mathcal{H}$ and $x.owner :=$ adv otherwise.

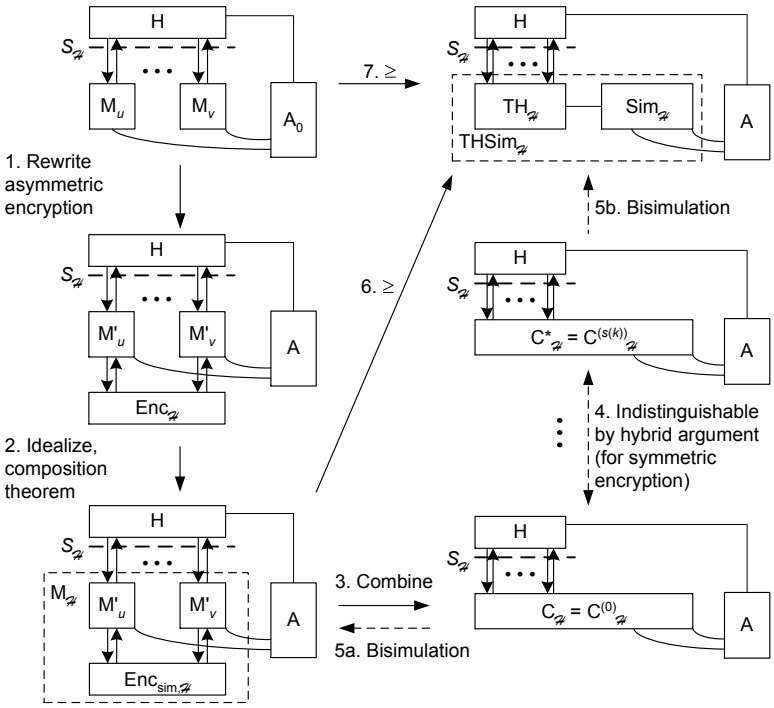

Figure 3: Proof with hybrid systems.

6.2.2 Hybrid Combined Systems

Two successive hybrid combined system differ only in the behavior for one symmetric encryption key $sk^{(i)}$: While $\mathsf{C}_{\mathcal{H}}^{(i)}$ still encrypts real messages with this key, $\mathsf{C}_{\mathcal{H}}^{(i+1)}$ encrypts simulated messages with it. The selection of $sk^{(i)}$ must guarantee that $sk^{(i)}$ is only encrypted with keys $sk^{(j)}$ for $j < i$, so that these encryptions have already been replaced by encryptions of fixed messages 1^{len^*}. We guarantee this by numbering the keys in the order in which they are first used for encryption. (The combined system has global knowledge of this.) This corresponds to the function order introduced for the definition of the NoComm property.

We define a hybrid combined system $\mathsf{C}_{\mathcal{H}}^{(i)}$ for every $i \in \mathbb{N}$. However, we will see that the number of different hybrid systems only grows polynomially in the security parameter. Each hybrid combined system $\mathsf{C}_{\mathcal{H}}^{(i)}$ keeps additional state compared with $\mathsf{C}_{\mathcal{H}}$.

- A global variable *used_keys* $\in \mathbb{N}$, initially set to 0. It counts how many honestly generated symmetric encryption keys have already been used for encryption.
- Each entry of type skse in D^* has two additional attributes: The Boolean attribute *used*, initially set to false, indicates whether the key has already been used for encryption. The attribute *pos* $\in \mathbb{N}$, initialized with $\downarrow$, indicates the position of this key in the order in which keys were first used for encryption.

The combined system $\mathsf{C}_{\mathcal{H}}^{(i)}$ processes commands like the initial combined system, except that the real words may be different when a ciphertext is generated or decrypted by an honest user. Hence only the constructor make_symenc and the decryption command sym_decrypt are affected, and only when the input sym_encrypt or sym_decrypt was made at a port $\mathsf{in}_u?$ for $u \in \mathcal{H}$. The local variable *sim* in the encryption constructor is set to true iff the key is used to encrypt simulated messages.

- *Symmetric encryption constructor:* $c^* \leftarrow \mathsf{make\_symenc}(sk^*, l)$ for $sk^*, l \in \{0,1\}^+$.
  Set $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $sk := sk^*[2]$, and $sr := sk^*[3]$. Let $skse^{\mathsf{ind}} := D^*[word = sk^*].ind$.
  if $D^*[skse^{\mathsf{ind}}].hnd_{\mathsf{a}} \neq \downarrow$ **then**
  $sim :=$ false

else
  if $D^*[skse^{\mathsf{ind}}].used = \mathsf{false}$ **then**
    $D^*[skse^{\mathsf{ind}}].used := \mathsf{true}$;
    $used\_keys := used\_keys + 1$;
    $D^*[skse^{\mathsf{ind}}].pos := used\_keys$
  end if
  $sim := (D^*[skse^{\mathsf{ind}}].pos \leq i)$
end if

If $sim = \mathsf{false}$ encrypt as $c \leftarrow \mathsf{sym\_encrypt}_{sk}((r,l))$ and otherwise as $c \leftarrow \mathsf{sym\_encrypt}_{sk}(1^{len^*})$ for $len^* := \mathsf{list\_len}(\mathsf{nonce\_len}(k), \mathsf{len}(l))$. Return $c^* := (\mathsf{symenc}, sr, r, c)$.

- *Symmetric decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{sym\_decrypt}(c^{\mathsf{hnd}}, skse^{\mathsf{hnd}})$.

  Parse c^{hnd} and $skse^{\mathsf{hnd}}$, and let $c^{\mathsf{ind}} := D^*[hnd_u = c^{\mathsf{hnd}}].ind$ and $sk^{\mathsf{ind}} := D^*[hnd_u = skse^{\mathsf{hnd}}].ind$. If $D^*[c^{\mathsf{ind}}].type \neq \mathsf{symenc}$ or $D^*[sk^{\mathsf{ind}}].type \neq \mathsf{skse}$, return $\downarrow$. If $D^*[sk^{\mathsf{ind}}].hnd_{\mathsf{a}} = \downarrow$ and $D[sk^{\mathsf{ind}}].used = \mathsf{false}$ then output $\downarrow$. Else let $(\mathsf{symenc}, sr, r, c) := D^*[c^{\mathsf{ind}}].word$ and $sk := D^*[sk^{\mathsf{ind}}].word[2]$.

  If $D^*[sk^{\mathsf{ind}}].hnd_{\mathsf{a}} \neq \downarrow$ or $D^*[sk^{\mathsf{ind}}].pos > i$ (a key for which we encrypt normally) then let $l^* := \mathsf{sym\_decrypt}_{sk}(c)$ and $l := l^*[2]$. If $sr \neq D^*[sk^{\mathsf{ind}}].word[3]$, or $l^* = \downarrow$, or $l^*[1] \neq r$, or if l is not a tagged list, set $l^{\mathsf{hnd}} := \downarrow$. Otherwise use l as the resulting word and compute and return l^{hnd} as in $\mathsf{C}_{\mathcal{H}}$.

  If $D^*[sk^{\mathsf{ind}}].pos \leq i$, let $((l_1, pkse_1), \ldots, (l_j, pkse_j)) := D^*[c^{\mathsf{ind}}].arg$. We claim that there exists a unique $j' \in \{1, \ldots, j\}$ such that $skse^{\mathsf{ind}} = pkse_{j'} + 1$. Output $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l_{j'})$.

Lemma 6.1 *The behavior of* $\mathsf{C}_{\mathcal{H}}^{(0)}$ *is equal to that of the initial combined system* $\mathsf{C}_{\mathcal{H}}$*. For every function* $s : \mathbb{N} \to \mathbb{N}$ *bounding the number of keys generated by honest users, in particular* $s(k) := n \cdot \mathsf{max\_in}(k)$*, the behavior of* $\mathsf{C}_{\mathcal{H}}^{(i)}$ *for the security parameter* k *and for* $i \geq s(k)$ *equals that of* $\mathsf{C}_{\mathcal{H}}^*$*.* □

Proof. This is clear since in $\mathsf{C}_{\mathcal{H}}^{(0)}$ we still treat all keys as in $\mathsf{C}_{\mathcal{H}}$, while for $i \geq s(k)$ we treat all keys as in $\mathsf{C}_{\mathcal{H}}^*$. ■

6.2.3 Low-level Combined Symmetric Encryption Machine

Within the hybrid argument, we do not want to argue individually with the secrecy and integrity of each ciphertext. We therefore first define a machine SymComb that corresponds almost precisely to the entire action of a hybrid system with one key. We then show that every successful attack against SymComb implies a successful attack on one of the machines SymDec and SymInt.

Definition 6.2 *(Machine* SymComb*) Given a symmetric encryption scheme, the machine* SymComb *is defined as follows: It has one input and one output port, a variable sk initialized with* $\downarrow$*, an initially empty database sym_ciphers with attributes* $(msg, ciph)$*, and the following transition rules:*

- *On input* (generate)*: If* $sk \neq \downarrow$*, then output* $\downarrow$*. Else generate a key as* $sk \leftarrow \mathsf{gen}_{\mathsf{SE}}(1^k)$ *and set* $b \xleftarrow{\mathcal{R}} \{0,1\}$*.*
- *On input* (symenc, m_0)*: If* $sk = \downarrow$*, then output* $\downarrow$*. Else set* $m_1 := 1^{\mathsf{len}(m_0)}$ *and* $c \leftarrow \mathsf{sym\_encrypt}_{sk}(m_b)$ *and* $sym\_ciphers :\Leftarrow (m_0, c)$*, and output* c*.*
- *On input* (symdec, c')*: If* $sk = \downarrow$*, then output* $\downarrow$*. Else if* $b = 0$ *return* $\mathsf{sym\_decrypt}_{sk}(c')$*; else return* $sym\_ciphers[ciph = c'].msg$*.*

The encryption scheme is called one-key reactively secure *if for every probabilistic polynomial-time machine* A_{SC} *that interacts with* SymComb *and finally outputs a bit* b^* *(meant as a guess at* b*), the probability of the event* $b^* = b$ *is bounded by* $1/2 + g(k)$ *for a negligible function* g*.* ◇

Lemma 6.2 *A secure symmetric encryption scheme in the sense of Definitions 5.1 and 5.2 is also one-key reactively secure.* □

This is a standard cryptographic reduction proof which we postpone to Appendix C.

6.2.4 The Hybrid Argument

We now show that the combined systems $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{C}^*_{\mathcal{H}}$ are indistinguishable. The overall structure of this hybrid argument is standard for the case of a polynomially growing number of hybrids. The special aspects of our usage of symmetric encryption come in when we treat the cases of how a secret key can and cannot occur in larger terms, and why the possible occurrences do no harm.

The core of the hybrid argument is to show how the encryption machine SymComb can be used to simulate either $\mathsf{C}^{(i)}_{\mathcal{H}}$ or $\mathsf{C}^{(i+1)}_{\mathcal{H}}$, depending on the bit b in SymComb. We call the rest of this simulation $\mathsf{C}'^{(i)}_{\mathcal{H}}$, i.e., the combination of $\mathsf{C}'^{(i)}_{\mathcal{H}}$ and SymComb should yield $\mathsf{C}^{(i)}_{\mathcal{H}}$ or $\mathsf{C}^{(i+1)}_{\mathcal{H}}$ depending on the bit b in SymComb. Clearly, we use SymComb for encryption and decryption with the i-th used key. The problem is what we do if the two hybrid systems (both or none) use the key in other operations. In spite of our assumptions this is not impossible, e.g., they may put it into a list and send it over a secure channel. Thus we let $\mathsf{C}'^{(i)}_{\mathcal{H}}$ choose its own key for these operations, independently of the key chosen in SymComb. The main task will be to show that this does not make the simulation distinguishable.

Definition 6.3 *The rewritten hybrid system $\mathsf{C}'^{(i)}_{\mathcal{H}}$ is defined exactly like $\mathsf{C}^{(i)}_{\mathcal{H}}$ with the following exceptions for inputs at $\mathsf{in}_u?$ with $u \in \mathcal{H}$:*

- *In the symmetric encryption constructor used in a command $c^{\mathsf{hnd}} \leftarrow \mathsf{sym\_encrypt}(skse^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ for the i-th used key, i.e., for $D^*[hnd_u = skse^{\mathsf{hnd}}].pos = i$, the algorithm $\mathsf{sym\_encrypt}_{sk}(\cdot)$ is replaced by calls to SymComb. Moreover, when this key first gets its attribute $pos := i$, then (generate) is input to SymComb.*
- *In symmetric decryption $l^{\mathsf{hnd}} \leftarrow \mathsf{sym\_decrypt}(skse^{\mathsf{hnd}}, c^{\mathsf{hnd}})$ for $D^*[hnd_u = skse^{\mathsf{hnd}}].pos = i$, the algorithm $\mathsf{sym\_decrypt}_{sk}(\cdot)$ is replaced by calls to SymComb.*

$\diamond$

Note that we have not replaced key generation in the definition of $\mathsf{C}'^{(i)}_{\mathcal{H}}$; hence we have a key sk^* in SymComb and another key $sk^{(i)} := D^*[hnd_u = skse^{\mathsf{hnd}}].word[2]$ in $\mathsf{C}'^{(i)}_{\mathcal{H}}$.

Lemma 6.3 *The combination of $\mathsf{C}'^{(i)}_{\mathcal{H}}$ and SymComb with bit $b = 0$ is reactively indistinguishable from $\mathsf{C}^{(i)}_{\mathcal{H}}$, and with bit $b = 1$ it is indistinguishable from $\mathsf{C}^{(i+1)}_{\mathcal{H}}$.* □

The proof is postponed to Appendix C. We now put all our lemmas together to show the following theorem about the main new proof parts for symmetric encryption.

Theorem 6.2 *Given a secure encryption scheme according to Definitions 5.1 and 5.2, the initial and final hybrid combined systems $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{C}^*_{\mathcal{H}}$ defined in Section 6.2.1 are reactively indistinguishable.* □

Proof. Assume for contradiction that there is a reactive distinguisher Dis that distinguishes $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{C}^*_{\mathcal{H}}$ with not negligible advantage $p(k)$. Here Dis combines honest users, adversary, and final distinguisher. Similar to Definition 2.1, the advantage is defined as $|q^*(k) - q(k)|$ where $q(k)$ and $q^*(k)$ denote the probabilities that Dis outputs 1 if it is run together with $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{C}^*_{\mathcal{H}}$, respectively, for the security parameter k.

Then we construct a successful adversary A_{SC} against the underlying symmetric encryption scheme, more precisely against the machine SymComb from Definition 6.2. This adversary A_{SC} is defined as follows: Given the security parameter k, it randomly chooses $i \xleftarrow{\mathcal{R}} \{0, \ldots s(k) - 1\}$, where s is the polynomial bound on the number of different hybrids from Lemma 6.1. Then it simulates the rewritten hybrid system $\mathsf{C}'^{(i)}_{\mathcal{H}}$ from Definition 6.3 in interaction with the reactive distinguisher Dis, where it lets $\mathsf{C}'^{(i)}_{\mathcal{H}}$ interact directly with the machine SymComb that A_{SC} attacks. If Dis outputs a bit b^*, then A_{SC} also outputs b^*.

By Lemma 6.3, the constructed adversary A_{SC} together with SymComb, and given a choice of i, perfectly simulates either $\mathsf{C}^{(i)}_{\mathcal{H}}$ and $\mathsf{C}^{(i+1)}_{\mathcal{H}}$, depending on the bit b in SymComb. Let q_i denote the probability that Dis outputs 1 if it is run together with $\mathsf{C}^{(i)}_{\mathcal{H}}$; we now omit the security parameter k for readability. The probability that A_{SC} guesses correctly for a specific i is $\frac{1}{2}(1 - q_i) + \frac{1}{2}(q_{i+1})$ (because for $b = 0$ we want Dis

to output $b^* = 0$). By Lemma 6.1, we have $q_0 = q$ and $q_s = q^*$. Hence the success probability of $\mathsf{A_{SC}}$, over the random choice of i from $\{0, \ldots, s-1\}$, is given by

$$\frac{1}{s}\sum_{i=0}^{s-1}\frac{1}{2}(1 + q_{i+1} - q_i) = \frac{1}{2} + \frac{1}{2s}(q^* - q).$$

As s is a polynomial, the absolute value of the difference between this guessing probability and $\frac{1}{2}$ is not negligible. If it is negative for almost all k, then we invert the output of $\mathsf{A_{SC}}$ to obtain an attacker with positive not negligible guessing advantage. This is the desired contradiction to Lemma 6.2.

■

6.2.5 The Bisimulations

Finally we have to show how the bisimulations of the original cryptographic library are extended for symmetric encryption. This corresponds to Steps 5a and 5b in Figure 3. The bisimulations are now mappings from $\mathsf{C}_\mathcal{H}$ to $\mathsf{M}_\mathcal{H}$ and from $\mathsf{C}^*_\mathcal{H}$ to $\mathsf{THSim}_\mathcal{H}$, called derivations in [4] because they essentially extract a part of the combined system again. As the initial and final combined systems both equal the original combined system on entries not belonging to symmetric encryption, these can both be extensions of the derivations from [4]. This is a tedious part of the proof without much novelty for symmetric encryption, hence we only sketch it.

The bisimulation proof relies on certain properties of the individual systems (ideal, real, and simulator), and on joint invariants of the combined systems. These are the lemmas in Sections 4-6 of [6] and the invariants in Section 7.2.4. Most of these properties are retained without change or adapted in obvious ways. Examples of retained properties are that indices and handles are unique, that length bounds are retained, and the equality of real and ideal lengths. An example of a property adapted in an obvious way is that real and abstract lengths are equal, except for a few types that do not correspond to real sendable words. Here the new type *pkse* is added to the exceptions.

Further, in order to show that the ideal look-up procedure for decryption works, we have to add an invariant that essentially covers the case that the simulator's action upon receipt of a new adversary key enters all possible encryptions, which it actually does in the procedure real2id_skse. More formally, the invariant, similar to one for symmetric authentication in [5], states that real parsing of an entry of type symenc *only* succeeds if the corresponding attributes are already present ideally.

The purpose of the derivations and the properties and invariants is described by the following definition.

Definition 6.4 *(Bisimulation Property) By "an input retains all invariants" we mean the following:*

- *The resulting transitions of $\mathsf{C}_\mathcal{H}$ and $\mathsf{C}^*_\mathcal{H}$ retain the invariants if they were true before the input.*
- *If the input is made to $\mathsf{M}_\mathcal{H}$ in the state derived from $\mathsf{C}_\mathcal{H}$, then the probability distribution of the next state equals that of the states derived from the next state of $\mathsf{C}_\mathcal{H}$. Similarly, If the input is made to $\mathsf{THSim}_\mathcal{H}$ in the state derived from $\mathsf{C}^*_\mathcal{H}$, then the probability distribution of the next state equals that of the states derived from the next state of $\mathsf{C}^*_\mathcal{H}$. This is called "correct derivation".*

◇

All the properties and invariants are obviously true initially when all databases are empty and the counters 0. Then we would like to show that indeed all inputs retain all invariants. Unfortunately, this is not true for *all* runs of the combined systems, e.g., if two nonces collide in the generation of two different secret keys. These exceptional runs are collected in *error sets*.

Hence the remaining part of a fully detailed proof consists of a relatively long and tedious part that shows that indeed all new inputs retain all invariants, except for certain well-defined error sets, and final reduction proofs that show that the overall probability of all error sets is negligible. This part is aided by certain lemmas from [6] that for each type of inputs (basic commands, send commands, and network inputs) a majority of the invariants is automatically fulfilled by general aspects of the cryptographic library. These lemmas continue to hold when symmetric encryption is added, which must be verified by text inspection.

The error sets that arise due to symmetric encryption are all of already known types, because the "main" cryptographic properties, both secrecy and ciphertext integrity, were already taken care of in the hybrid argument. In particular, it has to be shown that no two entries of nonces or keys made by honest participants collide, and that the adversary cannot guess random values and keys that he should ideally not be able to know. All these proofs work as in [6].

7 Conclusion

We have presented a provably secure idealization of symmetric encryption within the Dolev-Yao style cryptographic library from [4], which allows for cryptographically sound security proofs in an entirely abstract way accessible to current automated proof tools. Security holds under arbitrary attacks and in arbitrary contexts, and is based on the standard definition of authenticated encryption.

The benefit of adding symmetric encryption to the cryptographic library is impressive: Now 42 of the 50 protocols of the Clark-Jacob library can be expressed with the operations and constraints of the cryptographic library, while only 12 protocols could be expressed before. Among the remaining eight protocols, only one is excluded because of the commitment problem, five require hash functions (although one might already model some of them by message authentication codes), and two require number-theoretic operations like exponentiation and exclusive or.

References

[1] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 82–94, 2001.

[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 *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[3] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2003.

[4] 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/`.

[5] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2003. Extended version in IACR Cryptology ePrint Archive 2003/145, `http://eprint.iacr.org/`.

[6] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. `http://eprint.iacr.org/`.

[7] D. Beaver. Plug and play encryption. In *Advances in Cryptology: CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 1997.

[8] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology: EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1992.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology: ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[10] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[11] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient constructions. In *Advances in Cryptology: ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2000.

[12] J. Benaloh and D. Tuinstra. Uncoercible communication. Computer Science Technical Report TR-MCS-94-1, Clarkson University, 1994.

[13] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proc. 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 639–648, 1996.

[14] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 209–218, 1998.

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

[16] I. Damgård and J. B. Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *Advances in Cryptology: CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 432–450. Springer, 2000.

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

[18] R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 372–381, 2003.

[19] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. Manuscript, 2004.

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

[21] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proc. 5th ACM Conference on Computer and Communications Security*, pages 112–121, 1998.

[22] J. Millen. On the freedom of decryption. *Information Processing Letters*, 86(6):329–333, June 2003.

[23] J. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 725–733, 1998.

[24] J. Mitchell, M. Mitchell, A. Scedrov, and V. Teague. A probabilistic polynominal-time process calculus for analysis of cryptographic protocols (preliminary report). *Electronic Notes in Theoretical Computer Science*, 47:1–31, 2001.

[25] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001. Extended version in Cryptology ePrint Archive, Report 2000/066, `http://eprint.iacr.org/`.

[26] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 196–205, 2001.

[27] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

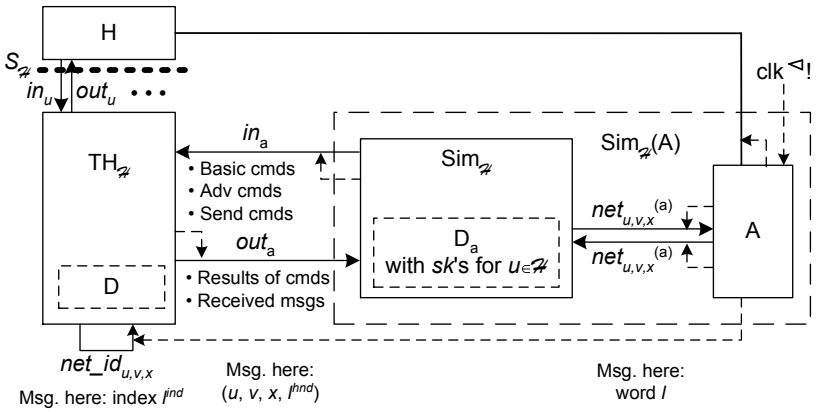

Figure 4: Set-up of the simulator.

A Simulator

We now give a formal description of the simulator $\mathsf{Sim}_{\mathcal{H}}$ sketched in Section 6.1.

A.1 States of the Simulator

The state of $\mathsf{Sim}_{\mathcal{H}}$ consists of a database D_{a} and variables $curhnd_{\mathsf{a}}$ and $steps_{\mathsf{p}?}$ for each input port $\mathsf{p}?$. Each entry in D_{a} has the following attributes:

- $x.hnd_{\mathsf{a}} \in \mathcal{HNDS}$ is used as the primary key attribute in D_{a}. However, its use is not as straightforward as in the ideal and real system, since entries are created by completely parsing an incoming message recursively.
- $x.word \in \{0,1\}^*$ is the real representation of x.
- $x.add\_arg$ is a list of additional arguments. Typically it is (). However, for our key identifiers it is (adv) if the corresponding secret key was received from the adversary, while for keys from honest users, where the simulator generated an encryption key, it is of the form (honest, sk^*).

The variable $curhnd_{\mathsf{a}}$ denotes the current size of D_{a}, except temporarily within an algorithm $\mathsf{id2real}$. The variables $steps_{\mathsf{p}?}$ count the inputs at each port. The corresponding bounds $bound_{\mathsf{p}?}$ are $\mathsf{max\_in}(k)$ for the network ports and $\mathsf{max\_in_a}(k)$ for $\mathsf{out_a}?$. These bounds are only included to ensure polynomial runtime, but in order to obtain the correct functionality, the second bound must not be reached as this would destroy the interaction of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$. This would allow for distinguishing the ideal and the real system. For our new primitive, we have to enlarge the second bound which does not alter the proof, as it remains polynomially bounded. Length functions for inputs are tacitly defined by the domains again.

A.2 Evaluation of Send Commands

When $\mathsf{Sim}_{\mathcal{H}}$ receives an "unsolicited" input from $\mathsf{TH}_{\mathcal{H}}$ (in contrast to the immediate result of a local command), this is the result $m = (u, v, \mathsf{i}, l^{\mathsf{hnd}})$ of a send command by an honest user (here for an insecure channel). $\mathsf{Sim}_{\mathcal{H}}$ looks up if it already has a corresponding real message $l := D_{\mathsf{a}}[l^{\mathsf{hnd}}].word$ and otherwise constructs it by an algorithm $l \leftarrow \mathsf{id2real}(l^{\mathsf{hnd}})$ (with side-effects). It outputs l at port $\mathsf{net}_{u,v,\mathsf{i}}!$.

The algorithm $\mathsf{id2real}$ is recursive; each layer builds up a real word given the real words for certain abstract components. We only need to add new type-dependent constructions for our new types, but we briefly repeat the overall structure to set the context.

1. Call $(type, (m_1^{\mathsf{hnd}}, \ldots, m_j^{\mathsf{hnd}})) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ (where we ignore some parentheses in the case $type = \mathsf{symenc}$) expecting $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ and $j \leq \mathsf{max\_len}(k)$, and

$m_i^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ if $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and otherwise $\mathsf{len}(m_i^{\mathsf{hnd}}) \leq \mathsf{max\_len}(k)$ (with certain domain expectations in the arguments m_i^{hnd} that are automatically fulfilled in interaction with $\mathsf{TH}_{\mathcal{H}}$, also for the now extended command adv_parse for the new types).

2. For $i := 1, \ldots, j$: If $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and $m_i^{\mathsf{hnd}} > curhnd_{\mathsf{a}}$, set $curhnd_{\mathsf{a}}$++.

3. For $i := 1, \ldots, j$: If $m_i^{\mathsf{hnd}} \notin \mathcal{HNDS}$, set $m_i := m_i^{\mathsf{hnd}}$. Else if $D_{\mathsf{a}}[m_i^{\mathsf{hnd}}] \neq \downarrow$, let $m_i := D_{\mathsf{a}}[m_i^{\mathsf{hnd}}].word$. Else make a recursive call $m_i \leftarrow \mathsf{id2real}(m_i^{\mathsf{hnd}})$. Let $arg^{\mathsf{real}} := (m_1, \ldots, m_j)$.

4. Construct and enter the real message m depending on *type*; here we only list the new types:

   - If $type = \mathsf{pkse}$, call $sk^* \leftarrow \mathsf{make\_symenc\_key}()$ and set $m := \epsilon$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, (\mathsf{honest}, sk^*))$.
   - If $type = \mathsf{skse}$, let $pkse^{\mathsf{hnd}} := m_1^{\mathsf{hnd}}$. We claim that $D_{\mathsf{a}}[pkse^{\mathsf{hnd}}].add\_arg$ is of the form (honest, sk^*). Set $m := sk^*$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.
   - If $type = \mathsf{symenc}$, we claim that $l^{\mathsf{hnd}} := m_1^{\mathsf{hnd}} \neq \downarrow$ and $pkse^{\mathsf{hnd}} := m_2^{\mathsf{hnd}} \neq \downarrow$, and distinguish two cases: If $D_{\mathsf{a}}[pkse^{\mathsf{hnd}}].add\_arg[1] = \mathsf{honest}$, let $sk^* := D_{\mathsf{a}}[pkse^{\mathsf{hnd}}].add\_arg[2]$, else $sk^* := D_{\mathsf{a}}[pkse^{\mathsf{hnd}} + 1].word$.

     If $l^{\mathsf{hnd}} \in \mathcal{HNDS}$ (i.e., a cleartext handle, not only a length was output), let $l := m_1$, $m \leftarrow \mathsf{make\_symenc}(sk^*, l)$, and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.

     Otherwise we claim that $len := l^{\mathsf{hnd}} \in \mathbb{N}$. Then $\mathsf{Sim}_{\mathcal{H}}$ encrypts a fixed message of the correct length; it must not be a list. Let $len^* := \mathsf{list\_len}(\mathsf{nonce\_len}(k), len)$, $sk := sk^*[2]$, and $sr := sk^*[3]$. Encrypt $c \leftarrow \mathsf{sym\_encrypt}_{sk}(1^{len^*})$ and set $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $m := (\mathsf{symenc}, sr, r, c)$, and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.

A.3 Evaluation of Network Inputs

When $\mathsf{Sim}_{\mathcal{H}}$ receives an input l from A at a port $\mathsf{net}_{w,u,\mathsf{i}}?$ with $\mathsf{len}(l) \leq \mathsf{max\_len}(k)$, it verifies that l is a tagged list. If yes, it translates l into a corresponding handle l^{hnd} by a recursive algorithm $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ (with side-effects), and outputs $\mathsf{adv\_send\_i}(w, u, l^{\mathsf{hnd}})$ at port $\mathsf{in_a}!$. The algorithm real2id recursively parses the real message, builds up a corresponding term in $\mathsf{TH}_{\mathcal{H}}$, and enters all messages into D_{a}.

For an arbitrary message $m \in \{0,1\}^+$, $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$ works as follows. If there is already a handle m^{hnd} with $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, it returns that. Else it sets $(type, arg) := \mathsf{parse}(m)$ and calls a type-specific algorithm $add\_arg \leftarrow \mathsf{real2id\_}type(m, arg)$. After this, real2id sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$. We have to provide the type-specific algorithms for our new types.

- $add\_arg \leftarrow \mathsf{real2id\_skse}(m, ())$. Call $skse^{\mathsf{hnd}} \leftarrow \mathsf{gen\_symenc\_key}()$ at $\mathsf{in_a}!$ and set $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}\mathsf{++}, \epsilon, (\mathsf{adv}))$ (for the key identifier), and $add\_arg = ()$ (for the secret key).

  Let $m =: (\mathsf{skse}, sk, sr)$; this format is ensured by the preceding parsing. For each handle c^{hnd} with $D_{\mathsf{a}}[c^{\mathsf{hnd}}].type = \mathsf{symenc}$ and $D_{\mathsf{a}}[c^{\mathsf{hnd}}].word = (\mathsf{symenc}, sr, r, c)$ for $r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, $c \in \{0,1\}^{\mathsf{symenc\_len}'(k, \mathsf{len}(l))}$, $\mathsf{sym\_decrypt}_{sk}(c) = (r, l)$ for some $l \in \{0,1\}^+$, make a recursive call $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ and call $v \leftarrow \mathsf{adv\_fix\_symenc\_content}(skse^{\mathsf{hnd}}, c^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Return add_arg.

- $add\_arg \leftarrow \mathsf{real2id\_symenc}(m, ())$. Let $(\mathsf{symenc}, sr, r, c) := m$; parsing ensures this format.

  For $l \in \{0,1\}^+$, let $Skse_l := \{skse^{\mathsf{hnd}} \mid D_{\mathsf{a}}[skse^{\mathsf{hnd}}].type = \mathsf{skse} \wedge D_{\mathsf{a}}[skse^{\mathsf{hnd}}].word[3] = sr \wedge \mathsf{sym\_decrypt}_{sk}(c) = (r, l)$ for $sk := D_{\mathsf{a}}[skse^{\mathsf{hnd}}].word[2]\}$ be the set of keys known to the adversary for which m decrypts to the message l. Let $Skse$ denote the union of the sets $Skse_l$.

  For each $Skse_l \neq \emptyset$ do the following: First, let $skse^{\mathsf{hnd}} \in Skse_l$ arbitrary and make a recursive call $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$. Secondly, call $c^{\mathsf{hnd}} \leftarrow \mathsf{sym\_encrypt}(skse^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Thirdly, for every $skse'^{\mathsf{hnd}} \in Skse_l \setminus \{skse^{\mathsf{hnd}}\}$ (in any order), call $v \leftarrow \mathsf{adv\_fix\_symenc\_content}(skse'^{\mathsf{hnd}}, c^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Return ().

  If $Skse = \emptyset$, call $c^{\mathsf{hnd}} \leftarrow \mathsf{adv\_unknown\_symenc}(\mathsf{len}(m))$ at $\mathsf{in_a}!$ and return ().

A.4 Properties of the Simulator

The simulator is polynomial-time. Further, no handle output by $\mathsf{TH}_{\mathcal{H}}$ is rejected by $\mathsf{Sim}_{\mathcal{H}}$, and the counters $steps_{\mathsf{out_a}?}$ of $\mathsf{Sim}_{\mathcal{H}}$ and $steps_{\mathsf{in_a}?}$ of $\mathsf{TH}_{\mathcal{H}}$ never reach their bounds. This is shown as in [4], except for the new bound $\mathsf{max\_in_a}$ for $steps_{\mathsf{in_a}?}$ and $steps_{\mathsf{out_a}?}$, cf. Section 4.2. Because of the interaction of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ in $\mathsf{real2id}$, these steps are increased linearly in the number of existing encryption and existing keys, since a new secret key might update the arguments of each existing encryption entry, and a new encryption can get any existing key as an argument. This means that we have to enlarge the bounds at $\mathsf{in_a}?$ and $\mathsf{out_a}?$ to maintain the correct functionality of the simulator. However, only a polynomial number of encryptions and keys can be created (a coarse bound is $n \cdot \mathsf{max\_in}(k)$ for entries of the honest users plus the polynomial runtime of A for the remaining ones). We omit further details.

B Corresponding Ideal Length Functions and Bounds

For given real length functions $\mathsf{list\_len}$, $\mathsf{nonce\_len}$, $\mathsf{skse\_len}$, and $\mathsf{symenc\_len}$, the corresponding ideal length functions are computed as follows:

- $\mathsf{skse\_len}^*(k) := \mathsf{list\_len}(\mathsf{len}(\mathsf{skse}), \mathsf{skse\_len}(k), \mathsf{nonce\_len}(k))$; this must be bounded by $\mathsf{max\_len}(k)$;
- $\mathsf{symenc\_len}'(k, l) := \mathsf{symenc\_len}(k, \mathsf{list\_len}(\mathsf{nonce\_len}(k), l))$;
- $\mathsf{symenc\_len}^*(k, l) := \mathsf{list\_len}(\mathsf{len}(\mathsf{symenc}), \mathsf{nonce\_len}(k), \mathsf{nonce\_len}(k), \mathsf{symenc\_len}'(k, l))$.

C Postponed Proofs

C.1 Proof of Lemma 6.2

Let $\mathsf{A_{SC}}$ be an adversary that succeeds in attacking $\mathsf{SymComb}$ with probability $\frac{1}{2} + p$ for a not negligible function p. We now construct an adversary $\mathsf{A_{SD}}$ against SymDec as follows. $\mathsf{A_{SD}}$ has the adversary $\mathsf{A_{SC}}$ as a blackbox submachine and maintains an initially empty database $sym\_ciphers_{SD}$ with attributes $(msg, ciph)$, both ranging over $\{0,1\}^+$, and a bit g, initially 0. We now defined how $\mathsf{A_{SD}}$ reacts on all outputs that $\mathsf{A_{SC}}$ makes (usually to $\mathsf{SymComb}$):

- $(\mathsf{generate})$. Here $\mathsf{A_{SD}}$ sets $g := 1$.
- (symenc, m_0). If $g = 0$, then $\mathsf{A_{SD}}$ returns $\downarrow$. Else it outputs $(\mathsf{symenc}, m_0, 1^{\mathsf{len}(m_0)})$ to SymDec, which answers with a ciphertext c. Then $\mathsf{A_{SD}}$ sets $sym\_ciphers_{SD} :\Leftarrow (m_0, c)$ and returns c to $\mathsf{A_{SC}}$.
- (symdec, c). If $g = 0$, then $\mathsf{A_{SD}}$ returns $\downarrow$. Else it sets $m := sym\_ciphers_{SD}[ciph = c].msg$. If $m \neq \downarrow$, it outputs m to $\mathsf{A_{SC}}$, otherwise it outputs (symdec, c) to SymDec and forwards the obtained message to $\mathsf{A_{SC}}$.
- A bit b^* as its guess of b. Then $\mathsf{A_{SD}}$ also outputs b^*.

We show that the adversary $\mathsf{A_{SD}}$ together with the machine SymDec perfectly simulates the machine $\mathsf{SymComb}$ with the bit b of SymDec unless the ciphertext integrity of the encryption scheme is violated. For this, we establish the following three invariants for runs of $\mathsf{A_{SD}}$ together with SymDec and runs of $\mathsf{SymComb}$ if they choose the same key sk and get the same inputs

1. The database $sym\_ciphers_{SD}$ of $\mathsf{A_{SD}}$ is always equal to the database $sym\_ciphers$ of $\mathsf{SymComb}$.
2. If $b = 0$ and $(m, c) \in sym\_ciphers_{SD}$, then $m = \mathsf{sym\_decrypt}_{sk}(c)$.
3. We have $(m, c) \in sym\_ciphers_{SD}$ for some m if and only if $c \in C$, the set of ciphertexts in SymDec.

We now show that the invariants are retained and the outputs of the simulation correct except in certain runs that violate ciphertext integrity. Before the first output (generate) of $\mathsf{A_{SC}}$, encryption and decryption commands to SymComb always yield $\downarrow$ because of $sk = \downarrow$ in SymComb, which is exactly what $\mathsf{A_{SD}}$ does. Hence assume in the following that an output (generate) already occurred, and thus $sk \neq \downarrow$ in SymComb. The simulation of encryption commands and further key generation commands is clearly perfect, and the invariants remain correct. Now we consider a decryption command (symdec, c). We set $m := sym\_ciphers_{SD}[ciph = c].msg$ and distinguish four cases.

- If $m \neq \downarrow$ and $b = 0$, then $\mathsf{A_{SD}}$ outputs m, while SymComb outputs $\mathsf{sym\_decrypt}_{sk}(c)$. This equals m by Invariant 2.
- If $m \neq \downarrow$ and $b = 1$, then $\mathsf{A_{SD}}$ outputs m, while SymComb outputs $sym\_ciphers[ciph = c].msg$. This equals m by Invariant 1.
- If $m = \downarrow$ and $b = 0$, then $\mathsf{A_{SD}}$ outputs (symdec, c) to SymDec. Invariant 3 implies that $c \notin C$. Hence both SymDec and SymComb output $\mathsf{sym\_decrypt}_{sk}(c)$.
- If $m = \downarrow$ and $b = 1$, then $\mathsf{A_{SD}}$ outputs (symdec, c) to SymDec. We again have $c \notin C$; hence SymDec outputs $m' = \mathsf{sym\_decrypt}_{sk}(c)$. SymComb returns $m^* := sym\_ciphers[ciph = c].msg$, where Invariant 1 implies $m^* = \downarrow$. Hence here we obtain the only exception to perfect simulation if $m' \neq \downarrow$.

Let q be the probability of the runs in which the only exception to perfect simulation (in the fourth case) occurs. Then the success probability of the adversary $\mathsf{A_{SD}}$ against SymDec is at least $\frac{1}{2} + p - q$, because in all other cases $\mathsf{A_{SD}}$ is successful if and only if $\mathsf{A_{SC}}$ is successful against SymComb. If $p - q$ is not negligible, we have obtained the desired contradiction to the given chosen-ciphertext security.

Otherwise q is not negligible. Then we derive a successful attack against ciphertext integrity. Intuitively this is possible because the ciphertext c in the exceptional case can be validly decrypted with sk although SymDec has never output c. Let $\mathsf{A_{SI}}$ be an adversary against SymInt that acts like $\mathsf{A_{SD}}$, but when $\mathsf{A_{SD}}$ outputs $(\mathsf{symenc}, m_0, 1^{\mathsf{len}(m_0)})$ to SymDec, then $\mathsf{A_{SI}}$ outputs $(\mathsf{symenc}, 1^{\mathsf{len}(m_0)})$ to SymInt. This clearly simulates the encryption commands perfectly for the case $b = 1$. For decryption and the case $m \neq \downarrow$ (and always $b = 1$), $\mathsf{A_{SD}}$ and thus $\mathsf{A_{SI}}$ both output m. If $m = \downarrow$, then $\mathsf{A_{SD}}$ and $\mathsf{A_{SI}}$ output (symdec, c) to SymDec and SymInt, respectively. Then SymInt always outputs $m' = \mathsf{sym\_decrypt}_{sk}(c)$, and we know that SymDec also outputs m' because $c \notin C$. Hence decryption is also simulated perfectly. Now an exception means $m' \neq \downarrow$, and $c \notin C$ is exactly the same condition as that for new ciphertexts in Definition 5.2. Hence in every exceptional run $\mathsf{A_{SI}}$ makes a successful attack against ciphertext integrity. Thus $\mathsf{A_{SI}}$ has success probability q against SymInt (even $2q$ because it always uses $b = 1$). This is the desired contradiction to the given ciphertext integrity.

C.2 Proof of Lemma 6.3

The lemma would clearly hold with perfect indistinguishability if the keys sk^* and $sk^{(i)}$ were equal, because then the use of the encryption machine SymComb instead of encrypting and decrypting oneself is a simple rewriting.

Hence it is sufficient to show that the use of $sk^{(i)}$ instead of sk^* in the operations other than en- and decryption is perfectly indistinguishable for the users and the adversary. For this we show that no information in the Shannon sense flows from the word $sk^{(i)} = D^*[skse^{\mathsf{ind}}].word[2]$ to the honest users and the adversary, except for the length of $sk^{(i)}$. Since sk^* and $sk^{(i)}$ are of the same length $\mathsf{skse\_len}(k)$, leaking the length of $sk^{(i)}$ does not destroy the perfect simulation. An overview of the cases in this proof is given in Figure 5.

We first show that no information about $sk^{(i)}$ is output at ports $\mathsf{out}_u!$ with $u \in \mathcal{H}$, i.e., to honest users. Such outputs occur as a result of basic commands and of network inputs. Most resulting outputs are database handles and types, which clearly do not reveal anything about the word attribute of $skse^{\mathsf{ind}}$. The only exceptions are the commands get_len, which outputs the length of an entry, and retrieve, which outputs the word attribute of an entry of type data. As explained above, leaking the length of $sk^{(i)}$ is no problem. Entries of type data can only be created by a command store, which does not depend on the word attribute of another entry, and in particular not on $sk^{(i)}$.

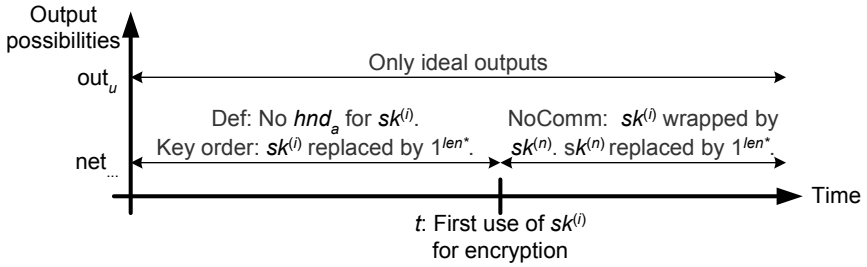

Figure 5: Absence of information flow from a simulated key $sk^{(i)}$.

We now show that no information flows at the network ports, i.e., to the adversary. We only need to consider authentic and insecure channels, and we distinguish outputs before and after the time t where $D^*[skse^{\mathsf{ind}}]$ is used for encryption for the first time. For the time until t, the definition of make_symenc in the hybrid systems guarantees that the adversary has no handle to this key, i.e., $t : D^*[skse^{\mathsf{ind}}].hnd_{\mathsf{a}} = \downarrow$ because keys with adversary handles are not counted. For the time after t, the property NoComm implies that in every term sent over any channel with $skse^{\mathsf{ind}}$ as a contained term, this term is wrapped by an encryption under a key $skse'^{\mathsf{ind}}$ for which the adversary does not have a handle. By definition of the commands adv_parse and sym_decrypt, this implies that the adversary cannot get a handle to $D^*[skse^{\mathsf{ind}}]$ after time t, and together with $t : D^*[skse^{\mathsf{ind}}].hnd_{\mathsf{a}} = \downarrow$ this implies $D^*[skse^{\mathsf{ind}}].hnd_{\mathsf{a}} = \downarrow$ also for the time after t.

We first show that no information about the key flows into database entries that ideally do not have this key as a component. For application data, nonces, and all types of keys, this is clear by definition. The word attribute of a list is fully determined by the word attributes of the contained terms of the list. The word attribute of a public-key encryption, digital signature, or authenticator is determined by the word attributes of the contained terms, a fresh random value, and on parts of the word attribute of the used secret key. For this secret key, we already showed that it does not depend on the word attributes of symmetric keys. Finally, the word attribute of symmetric encryptions also depends on word attributes of the contained terms, a fresh random value, and on parts of the word attribute of the used secret key, more precisely on $D^*[skse'^{\mathsf{ind}}].word[3]$ where $skse'^{\mathsf{ind}}$ is the index of the key used for the encryption. This part is independent of $sk^{(i)} = D^*[skse^{\mathsf{ind}}].word[2]$.

The case that an output term has $skse^{\mathsf{ind}}$ as an ideal component is the most interesting part of the proof: We only know that the adversary did not get a handle $D^*[skse^{\mathsf{ind}}].hnd_{\mathsf{a}}$ while the hybrid system prepared the real output. In the following we hence only treat such a term $l = D^*[l^{\mathsf{ind}}].word$ with $skse^{\mathsf{ind}} \in \mathsf{tree}(t : D[l^{\mathsf{ind}}])$. The initial combined system constructs network outputs like $\mathsf{Sim}_{\mathcal{H}}$, i.e., it translates the ideal output l^{ind} of $\mathsf{TH}_{\mathcal{H}}$ with the recursive procedure id2real. The interaction with $\mathsf{TH}_{\mathcal{H}}$ in this procedure is unchanged in all hybrid systems, and thus the term is parsed as far as possible with adv_parse. This gives an adversary handle to $skse^{\mathsf{ind}}$ except in certain cases, in particular that $skse^{\mathsf{ind}}$ is encrypted within this term. For these cases we nevertheless show the absence of information flow, using the prior replacements of real encryptions by encryptions of fixed messages in the hybrid system.

The first case is that $skse^{\mathsf{ind}}$ ideally occurs within a public-key encryption where the secret key is unknown to the adversary, i.e., as the cleartext argument or a component of that. But in the hybrid systems, all such real public-key encryptions are already replaced by encryptions of fixed messages 1^{len^*}, see Step 2 of Figure 3. Hence there is no information flow from the real cleartext word besides the length of the cleartext, which does not matter as shown above.

The second case is that $skse^{\mathsf{ind}}$ ideally occurs within a symmetric encryption with a key $skse'^{\mathsf{ind}}$ which has no adversary handle at the time t' where this term is sent. For the case $t' < t$ we know that $skse'^{\mathsf{ind}}$ was first used for encryption before time t and had no adversary handle then either. It thus got a position attribute $j := D^*[skse'^{\mathsf{ind}}].pos$ and we have $j < i$. For the case $t' > t$ the NoComm property ensures $t' : \mathsf{order}(D^*[skse'^{\mathsf{ind}}].ind) < t' : \mathsf{order}(D^*[skse^{\mathsf{ind}}].ind)$ which also implies that $skse'^{\mathsf{ind}}$ was first used for encryption before time t, that it had no adversary handle then either and thus also got a position attribute $j := D^*[skse'^{\mathsf{ind}}].pos$ with $j < i$. Thus all actual words encrypted with the corresponding real key $sk^{(j)}$ are simulated messages 1^{len^*}. In particular, they are independent of the real key $sk^{(i)}$, except possibly for the length, which does not matter as shown above.