A Universally Composable Cryptographic Library

Michael Backes, Birgit Pfitzmann, and Michael Waidner
IBM Zurich Research Lab

January 24, 2003

Abstract

Bridging the gap between formal methods and cryptography has recently received a lot of interest, i.e., investigating to what extent proofs of cryptographic protocols made with abstracted cryptographic operations are valid for real implementations. However, a major goal has not been achieved yet: a soundness proof for an abstract crypto-library as needed for the cryptographic protocols typically proved with formal methods, e.g., authentication and key exchange protocols. Prior work that directly justifies the typical Dolev-Yao abstraction is restricted to passive adversaries and certain protocol environments. Prior work starting from the cryptographic side entirely hides the cryptographic objects, so that the operations are not composable: While secure channels or signing of application data is modeled, one cannot encrypt a signature or sign a key.

We make the major step towards this goal: We specify an abstract crypto-library that allows composed operations, define a cryptographic realization, and prove that the abstraction is sound for arbitrary active attacks in arbitrary reactive scenarios. The library currently contains public-key encryption and signatures, nonces, lists, and application data.

The proof is a novel combination of a probabilistic, imperfect bisimulation with cryptographic reductions and static information-flow analysis.

1 Introduction

We present the first crypto-library with composable operations for which the gap between an abstract and a cryptographic version can be bridged for arbitrary active attacks, and within the context of arbitrary surrounding interactive protocols. This holds independently of the goals that one wants to prove about the surrounding protocols.

Why does one want to reconcile abstractions and cryptography? For about 20 years, formal and cryptographic methods for proving security protocols have coexisted, and each community was busy solving its own problems. For the formal-methods community, this main goal was to express security goals, and efficient proof techniques that allow full automation even for infinite state spaces. The abstraction from cryptography was essentially always the Dolev-Yao model. Here cryptographic operations, e.g., E for encryption and D for decryption, are considered as operators in a term algebra where only certain cancellation rules hold. For instance, encrypting a message m twice does not yield another message from the basic message space but the term $\mathsf{E}(\mathsf{E}(m))$. A typical cancellation rule is $\mathsf{D}(\mathsf{E}(m)) = m$ for all m. In the cryptographic community, the main goal was to define the security of many types of cryptographic algorithms and protocols against the most powerful active attacks, but within polynomial-time restrictions, and to prove that certain realizations, typically number-theoretic, offer this security under certain assumptions. However, the cryptographic definitions are at a first glance quite unrelated to the abstractions used in formal methods. On the other hand, for larger cryptographic protocols, the definitions, and even more the typical computational reduction proofs, are very long and error-prone, so that abstractions would really be useful.

Our work is the major step in combining the advantages of both approaches: For the first time we can show that an abstract library consisting of Dolev-Yao-style primitives is safely realized by a cryptographic implementation under arbitrary active attacks, in arbitrary protocol contexts, and for arbitrary security properties, in particular confidentiality, authenticity, and other integrity properties. The library contains public-key encryption and digital signatures, nonces, list operations, and arbitrary application data. Its real version is built from arbitrary, cryptographically secure public-key encryption and signature systems, enhanced by additional operations like tagging and randomization. The additional operations do not cost much efficiency compared with the underlying schemes. Thus protocol design via the abstraction does not lead to efficiency problems in the resulting protocols.

To model the active attacks, and also surrounding interactive protocols about which the protocol designer knows nothing a priori, we work in a reactive setting. Thus "our Dolev-Yao model" is not just an algebra, but a stateful system with possible operations for honest participants and adversaries, depending on prior cryptographic operations and network actions. In this sense it corresponds more to "the CSP Dolev-Yao model" or "the Strand-space Dolev-Yao model" etc., which also have models of participants, interaction, and adversary besides the underlying cryptographic term algebras.

The notion that the real crypto-library is "as good as" the abstract one in all contexts and for all security goals is given by simulatability. Briefly this means that everything that can happen to users of the real crypto-library, in the presence of arbitrary polynomial-time attackers that can corrupt their machines, modify messages on insecure channels, influence the scheduling, and influence the users themselves, can also happen to the same users with the abstract crypto-library, where attacks are much more restricted. Existing composition and preservation theorems for simulatability give us exactly the properties we want: First, if we design a protocol with the abstract crypto-library, and then plug the real crypto-library in, the entire real protocol is as good as the abstract one in the same simulatability sense. Secondly, if we prove specific security properties for the abstract protocol, they also hold for the real protocol.

Related work The Dolev-Yao model is from [16]. Cryptographic definitions of public-key encryption and signature schemes were developed in [38, 20, 15, 21]. First automated proof tools based on the Dolev-Yao model were presented in [30, 28, 24]. Early examples of the large body of work of embedding the Dolev-Yao model in standard tools are [1, 27]. Simulatability, the basic notion for comparing an abstract and a real system, was first invented for multi-party function evaluation [37, 19, 7, 29, 9], i.e., systems with only one initial input set and only one output set. An extension to a reactive scenario, where participants can make new inputs many times, e.g., start new sessions like key exchanges, was first fully defined in [33], with extensions to asynchronous systems in [34, 10]. We use the IO-automata model of Pfitzmann/Waidner. First examples of tool-supported proofs in that model exists [6, 5], using PVS [31]. Earlier, but less complete or less abstract reactive simulatability notions exist in [26, 23].

The reactive simulatability papers already had the goal of bridging the gap between abstract and concrete cryptographic systems. However, so far they abstracted from cryptography so completely that cryptographic operations cannot be composed. For instance, [34] models secure channels. While these are an important high-level security primitive also in the formal-methods literature, one cannot express many lower-level protocols with them alone. Models for low-level systems like signatures exist too, e.g., in [10], but those simply store what application data has already been signed, without the possibility to sign a signed message again or a key. One cannot simply extend this by allowing the export of keys and signatures as application data because that would immediately destroy the simulatability.

Another approach at bridging the abstraction gap started with the full Dolev-Yao model with composable algorithms, at least for symmetric encryption and synchronous protocols [3, 2]. There, however, the adversary is restricted to passive eavesdropping. Consequently, it was not necessary to choose a reactive model of a system and its honest users, and the security goals are all formulated as indistinguishability, i.e., if two abstract systems are indistinguishable by passive adversaries, then so are two corresponding real systems. There are no theorems about composition or property-preservation from the abstract system to the real system yet in this model. We pay for the greater applicability of reactive simulatability and for allowing active attacks by a much more complicated proof.

Some further approaches for special security goals or primitives are [36, 22, 25].

2 Overview of the Simulatable Cryptographic Library

In this section, we give an overview of our abstract and real crypto-library, in particular their relations to the standard Dolev-Yao model and standard cryptographic definitions, respectively. To understand some design decisions, a basic understanding of simulatability is important. Thus we first briefly sketch the definitions from [34].

2.1 Overview of Simulatability

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*. In a *standard cryptographic system*, the structures are derived from one intended structure and a trust model. The trust model consists 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. Each structure is complemented to a *configuration* by adding 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 general scheduling model in [34] 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 when it makes a transition. In real asynchronous cryptographic systems, network connections are typically scheduled by A. Thus a configuration is a runnable system, i.e., one gets a probability space of runs. Views of individual machines in these runs are the restriction to all in- and outputs this machine sees and its internal state.

Simulatability essentially means that whatever can happen to certain users in the real system can also happen to the same users in the ideal (abstract) system: For every structure $struc_1$ of the real system, every user H, and every adversary A_1 there exists an adversary A_2 on a corresponding ideal structure, $struc_2$, such that the view of H in the two configurations is indistinguishable. Indistinguishability is a well-defined cryptographic notion from [38]. This is shown in Figure 1. For

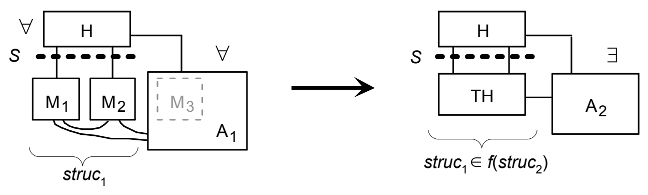

Figure 1: Overview of the simulatability definition. The view of H must be indistinguishable in the two configurations. In this example, $\mathcal{H} = \{1, 2\}$.

the crypto-library, we even show blackbox simulatability, where A_2 consists of a simulator Sim depending only on $struc_1$ and using A_1 as a blackbox submachine.

In a typical ideal system, each structure contains only one machine TH called *trusted host.* Corresponding structures are designated by a function f. For standard cryptographic systems, f maps a real structure to a trusted host with the same set S of specified ports, so that the same honest users can connect.

2.2 Overview of the Ideal System

An ideal (abstract) crypto-library 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 semantics. In a reactive scenario, this semantics is based on state, e.g., of who already knows which terms. We store state in a "database". Each entry 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. Thus the database index and these handles serve as an infinite, but efficiently constructible supply of global and local names for cryptographic objects. However, most libraries have export operations and leave message transport to their users ("token-based"). An actual implementation of the simulatable library might internally also be structured like this, but higher protocols are only automatically secure if they do not use this export function except via the special send operations.

The ideal crypto-library does not allow cheating. 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. Recall that our model is an entire reactive system like "the CSP Dolev-Yao model" and therefore contains an abstract network model. We offer three types of send commands, corresponding to three channel types $\{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$, meaning secure, authentic (but not private), and insecure. The types could be extended.

As to the operations, the main differences between our ideal crypto-library and the standard Dolev-Yao model are the following. Some of them already exist in prior extensions of the Dolev-Yao model.

- Signature schemes are not "inverses" of encryption schemes.
- Secure encryption schemes are necessarily probabilistic, and typically so are secure signature schemes. Hence if the same message is signed or encrypted

several times, we have to distinguish the versions, i.e., we make different database entries. For each entry, we only need to store its 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-like term.

- Secure signature schemes often have memory. The standard definition [21] does not even exclude that one signature divulges the entire history of messages signed before. We have to restrict this definition, but will allow a signature to divulge the number of previously signed messages, so that we include the most efficient provably secure schemes under classical assumptions like hardness of factoring [21, 11, 12].[1]
- We cannot (easily) allow participants to send secret keys around because then the simulation is not always possible.[2] Fortunately, for public-key cryptosystems this is not needed in standard protocols.
- Encryption schemes cannot keep the length of arbitrary cleartexts entirely secret. Typically one can even see the length quite precisely because message expansion is minimized. Hence we also allow this in the ideal system. A fixed-length version would be an easy addition to the library, or can be implemented on top of the library by padding to a fixed length.
- Adversaries may include incorrect messages in encrypted parts of a message which the current recipient cannot decrypt, but may possibly forward to another recipient who can, and will thus notice the incorrect format. Hence we also allow certain "garbage" terms in the ideal system.

Moreover, our ideal crypto-library contains explicit bounds on message lengths and the number of accepted inputs, so that a polynomial-time real system can simulate it correctly. In practice one would typically choose these bounds so large that they are not reached under ordinary circumstances.

2.3 Overview of the Real System

The real crypto-library offers its users the same commands as the ideal one, i.e., honest users operate on cryptographic objects via handles. This is actually quite close to standard APIs for real implementations of crypto-libraries that include key storage. The database of the real system contains real cryptographic keys, ciphertexts, etc., and sending a term on an insecure channel releases the actual bitstring to the adversary, who can do with it what he likes. The adversary can also insert arbitrary bitstrings on non-authentic channels. 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 base the implementation of the commands on arbitrary secure encryption and signature systems according to standard cryptographic definitions. However, we "idealize" the cryptographic objects and operations by measures similar to robust protocol design [4]. Some of these measures have already been used in prior, specialized designs.

- All objects are tagged with a type field so that, e.g., signatures cannot also be acceptable ciphertexts or keys.

[1]Memory-less schemes exist with either lower efficiency or based on stronger assumptions (e.g., [18, 14, 17]). We could add them to the library as an additional primitive.

[2]The primitives become "committing". This is well-known from individual simulation proofs. It is also the reason why the approach in [3], developed for symmetric cryptosystems, is hard to extend to active attacks.

- Several objects are also tagged with their parameters, e.g., signatures with the public key used.
- Randomized operations must be randomized completely. For instance, as the ideal system represents several signatures under the same message with the same key as different, the real system has to guarantee that they *will* be different, except for small error probabilities. Even probabilistic encryptions are randomized additionally because they are not always sufficiently random for keys chosen by the adversary.

The reason to tag signatures with the public key needed to verify them is that the usual definition of a secure signature scheme does not exclude "signature stealing:" Let (sk_h, pk_h) denote the key pair of a correct participant. With ordinary signatures an adversary might be able to compute a valid key pair (sk_a, pk_a) such that signatures that pass the test with pk_h also pass the test with pk_a. Thus, if a correct participant receives an encrypted signature on m, it might accept m as being signed by the adversary, although the adversary never saw m. It is easy to see that this would result in protocols that could not be simulated. (An example is given in [32], Section 6.4.) Our modification prevents this anomaly.

For the additional randomization of signatures, we include a random string r in the message to be signed. Alternatively we could replace r by a counter, and if a signature scheme is strongly randomized already we could omit r.

Ciphertexts are randomized by including the same random string r in the message to be encrypted and in the ciphertext. The outer r prevents collisions among ciphertexts from honest participants, the inner r ensures continued non-malleability.

3 Notation

We mostly use a straight font for $\mathsf{machines}$, $\mathsf{algorithms}$, $\mathsf{functions}$, and $\mathsf{constants}$; italics for *sets* and other *variables*, and a calligraphic font for $\mathcal{TYPES}$ and $\mathcal{I}$ndex sets.

We write ":=" for deterministic and "$\leftarrow$" for probabilistic assignment, and "$\xleftarrow{\mathcal{R}}$" for uniform random choice from a set. In an algorithm X executed by a machine M and with a named output m, "M returns n" means that M outputs $m := n$ and stops executing X. We usually omit the return statement if an execution path ends by returning a variable m as the output m. By $x := y{++}$ for integer variables x, y we mean $y := y + 1; x := y$.

A function $g : \mathbb{N} \to \mathbb{R}$ is called negligible, written $g \in NEGL$, if for all positive polynomials Q, $\exists k_0 \forall k \geq k_0$: $|g(k)| \leq 1/Q(k)$.

We make the following assumptions about the representation of structured data types.

- For reasons of distinguishability, we use a type system consisting of abstract datatypes $\mathcal{NAT}$ ("naturals"), $\mathcal{HNDS}$ ("handles"), $\mathcal{INDS}$ ("indices"), $\mathcal{BOOL}$ ("Boolean"), $\mathcal{ERR}$ ("error"), $\mathcal{CHARSTR}$ ("character strings"), $\mathcal{BITSTR}$ ("bitstrings"), and $\mathcal{LIST}$ ("lists"). The union of all these datatypes is denoted as $\mathcal{TYPES}$.
- The types $\mathcal{NAT}$, $\mathcal{HNDS}$, and $\mathcal{INDS}$ are isomorphic to $\mathbb{N} = \{1, 2, \ldots\}$, and we write elements and operations as for natural numbers. $\mathcal{NAT}$ is extended to $\mathcal{NAT}_0$, which is isomorphic to $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.
- We write $\mathcal{BOOL} = \{\mathsf{true}, \mathsf{false}\}$ and $\mathcal{ERR} = \{\downarrow\}$. $\mathcal{CHARSTR}$ is isomorphic to Σ^+ for some finite alphabet $\Sigma \supseteq \{a, \ldots, z\} \cup \{0, 1\}$ with $\downarrow \notin \Sigma$, and $\mathcal{BITSTR}$ isomorphic to $\{0, 1\}^*$.

- The type $\mathcal{LIST}$ contains lists of elements of $\{0,1\}^*$. The empty list is written $()$. We write a list of $x_1, \ldots, x_h$ as $(x_1, \ldots, x_j)$. If $l = (x_1, \ldots, x_j)$ then $l[i] := x_i$ for $1 \leq i \leq j$, and $l[i] := \downarrow$ for $i > j$. By "adding an element to a list" and similar formulations we mean appending it at the end.
- The ranges and domains of all algorithms and functions implicitly contain the error element $\downarrow$ of type $\mathcal{ERR}$. If $\downarrow$ is used as an argument for an algorithm, function or as a selector for a datatype (e.g., as an index of a list element), the result is always $\downarrow$. However, comparisons "if $v = \downarrow$" return true or false.
- In order to built up lists of elements of our abstract datatypes, and to apply cryptographic operations to them (again with the exception of $\downarrow$), we assume that elements of $\mathcal{TYPES} \setminus \mathcal{ERR}$ are uniquely encoded into the set $\{0,1\}^*$. The exact encoding does not matter, but if a function is efficiently computable for an abstract datatype, it must be efficiently computable also on the encoding. In particular, this comprises element retrieval for lists, comparisons and algorithms on natural numbers etc.
- The length $|w|$ of $w \in \{0,1\}^*$ is, as usual, the unique k with $w \in \{0,1\}^k$. The length $|w|$ of $w \in \mathcal{TYPES} \setminus \mathcal{ERR}$ for a given encoding en is defined as $|w| := |en(w)|$. We require the length of the encoding of a list l to depend only on the length of its encoded arguments: there is a function $\mathsf{list\_len}$ such that $|(x_1, \ldots, x_j)| = \mathsf{list\_len}(x_1, \ldots, x_j) \geq |x_1| + \ldots + |x_j|$ for all $j \in \mathbb{N}_0$ and all $x_1, \ldots, x_j \in \mathcal{TYPES} \setminus \mathcal{ERR}$.
- By a "database" D we mean a finite set of mappings from finite subsets of $\mathcal{CHARSTR}$ to $\mathcal{TYPES} \cup \{0,1\}^*$. These mappings are called *tuples* and their arguments are called *attributes*. With $t.a$ we mean the result of $t \in D$ on argument $a \in \Sigma^*$. As usual, $t.a = \downarrow$ means that $t.a$ is undefined.

  We use $D :\Leftarrow t$ as abbreviation of $D := D \cup \{t\}$. We often write tuples as lists of assignments, like $(a_1 := x_1, \ldots, a_k := x_k)$, and if the attributes and their order are clear from the context we just write $(x_1, \ldots, x_k)$.

  We will use some relational-database notation:

  - If $P(A)$ is a logical condition with the attributes in A as free variables then we define the selection $\sigma_{P(A)}(D)$ as the set of all $t \in D$ whose actual attribute values satisfy $P(A)$. Let $D[P(A)] := t$ if $\sigma_{P(A)}(D) = \{t\}$, and $D[P(A)] := \downarrow$ if $|\sigma_{P(A)}(D)| \neq 1$.
  - A *key attribute* is an attribute a such that $t.a$ is defined ($\neq \downarrow$) and unique for all $t \in D$. If we designate a specific key attribute a as a *primary key attribute* then we write $D[x]$ instead of $D[a = x]$.

4 Ideal System for a Cryptographic Library

We now present the ideal crypto-library. We include all definition details like ports and structures as needed for the notion of simulatability, because the ideal system is the abstract cryptographic module based on which protocols should be proved in future work. Hence it has to be defined precisely, and encoded faithfully into proof tools.

4.1 Structures

Given a number n of participants and a tuple L of parameters (mainly about lengths) discussed in Section 4.2, the ideal system is of the form

$$Sys_{n,L}^{\mathsf{cry,id}} = \{(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\},$$

where $\mathcal{H}$ denotes the set of honest participants (i.e., the access structure makes no restriction on the possible corruptions).

Actually, we consider two versions $Sys_{n,L}^{\mathsf{cry,id,stan}}$ and $Sys_{n,L}^{\mathsf{cry,id,loc}}$ called stand-alone and localized; in the former, in- and outputs for the users are scheduled by the adversary, in the latter locally (i.e., the crypto library is then used as local subroutines by protocols using it). The ports of $\mathsf{TH}_{\mathcal{H}}$ intended for the users are, respectively,

$$userports_{\mathcal{H}}^{\mathsf{stan}} := \{\mathsf{in}_u?, \mathsf{out}_u! \mid u \in \mathcal{H}\};$$

$$userports_{\mathcal{H}}^{\mathsf{loc}} := \{\mathsf{in}_u?, \mathsf{out}_u!, \mathsf{out}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}.$$

Intuitively each u represents one user. The ports of the users which connect to those ports are, respectively,

$$S_{\mathcal{H}}^{\mathsf{stan}^c} := \{\mathsf{in}_u!, \mathsf{out}_u? \mid u \in \mathcal{H}\};$$

$$S_{\mathcal{H}}^{\mathsf{loc}^c} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \mathcal{H}\}.$$

The localized version is shown in Figure 2.[3]

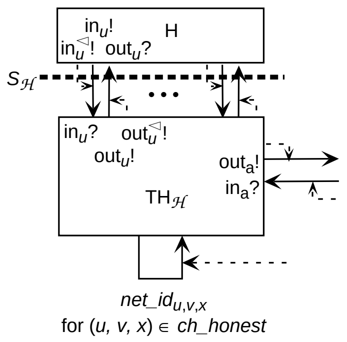

Figure 2: Ports of $\mathsf{TH}_{\mathcal{H}}$; localized version. Dotted arrows are clock connections; they end at buffers, which are not shown.

For the adversary, $\mathsf{TH}_{\mathcal{H}}$ mainly offers two ports $\mathsf{in_a}?$ and $\mathsf{out_a}!$. Furthermore, to allow the adversary to schedule messages even on secure channels, there are abstract versions of these secure channels. We use the following abbreviations:

$$\begin{aligned}
ch\_honest &:= \{(u,v,x) | u, v \in \mathcal{H} \wedge x \in \{\mathsf{s}, \mathsf{a}\}\};\\
ch\_from\_adv &:= \{(u,v,x) | v \in \mathcal{H} \wedge (u \notin \mathcal{H} \vee x = \mathsf{i})\};\\
ch\_to\_adv &:= \{(u,v,x) | u \in \mathcal{H} \wedge (v \notin \mathcal{H} \vee x \in \{\mathsf{a}, \mathsf{i}\})\}.
\end{aligned}$$

Then for $xtin\{\mathsf{stan}, \mathsf{loc}\}$:

$$\begin{aligned}
Ports_{\mathsf{TH}_{\mathcal{H}}^t} := \; & userports_{\mathcal{H}}^t \cup \{\mathsf{in_a}?, \mathsf{out_a}!, \mathsf{out_a}{}^{\triangleleft}!\}\\
& \cup \{\mathsf{net\_id}_{u,v,x}!, \mathsf{net\_id}_{u,v,x}? \mid (u,v,x) \in ch\_honest\}.
\end{aligned}$$

4.2 Lengths and Bounds

We already mentioned that we do not assume encryption systems to hide the length of the message. Furthermore, higher protocols may need to know the length of certain terms even for honest participants. Thus the trusted host is parameterized with

[3]The set $S_{\mathcal{H}}$ of specified ports is defined as the free ports at the end of the buffers. Hence here it is easier to show its complement, $S_{\mathcal{H}}^c$, which are the ports that H may have to connect to $\mathsf{TH}_{\mathcal{H}}$.

certain length functions denoting the length of a corresponding value in the real system. In abbreviated notation, these functions are $\mathsf{data\_len}^*(l)$, $\mathsf{list\_len}^*(l_1, \ldots, l_j)$ for all $j \in \mathbb{N}_0$, $\mathsf{nonce\_len}^*(k)$, $\mathsf{pks\_len}^*(k)$, $\mathsf{sig\_len}^*(k, l)$, $\mathsf{pke\_len}^*(k)$, and $\mathsf{enc\_len}^*(k, l)$. The domain of each parameter is $\mathbb{N}_0$, of each result $\mathbb{N}$, and k denotes the security parameter, while l and the l_i's are lengths of messages.

Furthermore, to allow a polynomial-time system to be as secure as this ideal system, we use functions $\mathsf{max\_len}(k)$, $\mathsf{max\_skc}(k)$, and $\mathsf{max\_in}(k)$ bounding the length of messages, the number of signatures per key, and the number of inputs at each port of correct real machines, respectively. To avoid unrealistic special cases, we require for all $k \in \mathbb{N}$ that $\mathsf{max\_len}(k) > 3$; $\mathsf{nonce\_len}^*(k), \mathsf{pks\_len}^*(k), \mathsf{pke\_len}^*(k) < \mathsf{max\_len}(k)$; $\mathsf{max\_skc}(k) \leq \mathsf{max\_in}(k)$; and $|i| < \mathsf{max\_len}(k)$ for all $i \in \mathbb{N}$ with $i \leq \mathsf{max\_skc}(k)$.

The tuple of all these functions is the system parameter L. Each function must be bounded by a (multivariate) polynomial and efficiently computable.

4.3 States

The state of $\mathsf{TH}_{\mathcal{H}}$ consists of a database D and variables *size*, $curhnd_u$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$, and $steps_{\mathsf{p?}}$ for each input port p?.

4.3.1 Database D

The main data structure of $\mathsf{TH}_{\mathcal{H}}$ is a database D. The entries of D are abstract representations of the data produced during a system run, together with the information on who knows these data. Each entry in D has attributes

$$(ind, type, arg, hnd_{u_1}, \ldots, hnd_{u_m}, hnd_{\mathsf{a}}, len)$$

where $\mathcal{H} = \{u_1, \ldots, u_m\}$. For each tuple $x \in D$:

- $x.ind \in \mathcal{INDS}$, called index, consecutively numbers all entries in D. We use it as a primary key attribute, i.e., we will write $D[i]$ for $D[ind = i]$.

- $x.type \in typeset := \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}, \mathsf{ske}, \mathsf{pke}, \mathsf{enc}, \mathsf{sks}, \mathsf{pks}, \mathsf{sig}, \mathsf{garbage}\}$ identifies the type of x. Further extensions of the library can extend this set.

- $x.arg = (a_1, a_2, \ldots, a_j)$ 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."[4] The argument domains are defined implicitly by the commands, see Section 4.5. They are easy to look up there because most types of entry are only constructed with on command; exceptions are explicitly mentioned there.

- $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ identifies how u knows this entry. The value a represents the adversary, and $x.hnd_u = \downarrow$ indicates that u does not know this entry. A value $x.hnd_u \neq \downarrow$ is called the *handle* for u to entry x. We always use a superscript "hnd" for handles and usually denote a handle to an entry $D[i]$ by i^{hnd}.

- $x.len \in \mathbb{N}_0$ denotes the "length" of the entry.[5]

[4]This is similar to the Dolev-Yao model: The arguments are just listed, not evaluated as in a real system.

[5]It is computed by $\mathsf{TH}_{\mathcal{H}}$ using the given length functions. For most types, this length could be recomputed when it is needed instead of storing it, but not for the type $\mathsf{garbage}$.

4.3.2 Conventions about Indices

Initially, D is empty. $\mathsf{TH}_{\mathcal{H}}$ keeps a variable $size$ of type $\mathcal{INDS}$ denoting the current number of elements in D. New entries x always receive $x.ind := size$++ (increment, then assign), and $x.ind$ is never changed. Thus, ind is indeed a key attribute.

4.3.3 Conventions about Handles

For each $u \in \mathcal{H} \cup \{\mathsf{a}\}$, $\mathsf{TH}_{\mathcal{H}}$ maintains a counter $curhnd_u$ (current handle) of type $\mathcal{HNDS}$ initialized with 0, and each new handle for u will be chosen as $i^{\mathsf{hnd}} := curhnd_u$++. Thus handles are uniquely and consecutively assigned, i.e., D always fulfills the invariant that $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 1$ for each $i^{\mathsf{hnd}} \leq curhnd_u$, and $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 0$ for all $i^{\mathsf{hnd}} > curhnd_u$. We call this fact "handle uniqueness."

The algorithm $\mathsf{ind2hnd}_u$ (with side effect) denotes that $\mathsf{TH}_{\mathcal{H}}$ determines a handle i^{hnd} for u to an entry $D[i]$: On input $i \in \mathcal{INDS}$, it returns $D[i].hnd_u$ if it is not $\downarrow$; else it sets $D[i].hnd_u := curhnd_u$++ and returns $D[i].hnd_u$. We extend it by $\mathsf{ind2hnd}_u(i) := i$ for $i \notin \mathcal{INDS}$. For lists, the algorithm $\mathsf{ind2hnd}_u^*$ applies $\mathsf{ind2hnd}_u$ to each element.

4.3.4 Input Counters

For each of its input ports $\mathsf{p}?$, $\mathsf{TH}_{\mathcal{H}}$ maintains a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ initialized with 0 for the number of inputs at that port. These counters are used to ensure polynomial runtime of $\mathsf{TH}_{\mathcal{H}}$. Hence for each port, there is a bound $bound_{\mathsf{p}?}$ that the counter must not exceed.[6] This bound is $\mathsf{max\_in}(k)$ for all ports except for $\mathsf{in_a}?$, where it can be $\mathsf{max\_in_a}(k) := 10n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ or any larger polynomial. (This fits a real system with the bound $\mathsf{max\_in}(k)$ for all ports.)

4.4 Overview of Possible Inputs

Each input c at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$ should be a list $(cmd, x_1, \ldots, x_j)$. We usually write it

$$y \leftarrow cmd(x_1, \ldots, x_j)$$

with a variable y designating the result that $\mathsf{TH}_{\mathcal{H}}$ returns at $\mathsf{out}_u!$. The value cmd should be a command string: Let

$$basic\_cmds := \{\mathsf{get\_type}, \mathsf{get\_len}, \mathsf{store}, \mathsf{retrieve}, \mathsf{list}, \mathsf{list\_proj}, \mathsf{gen\_nonce}, \mathsf{gen\_sig\_keypair}, \mathsf{sign}, \mathsf{verify}, \mathsf{pk\_of\_sig}, \mathsf{msg\_of\_sig}, \mathsf{gen\_enc\_keypair}, \mathsf{encrypt}, \mathsf{decrypt}, \mathsf{pk\_of\_enc}\};$$

$$send\_cmds := \{\mathsf{send\_s}, \mathsf{send\_a}, \mathsf{send\_i}\};$$

$$adv\_local\_cmds := \{\mathsf{adv\_garbage}, \mathsf{adv\_invalid\_ciph}, \mathsf{adv\_transform\_sig}, \mathsf{adv\_parse}\};$$

$$adv\_send\_cmds := \{\mathsf{adv\_send\_s}, \mathsf{adv\_send\_a}, \mathsf{adv\_send\_i}\}.$$

Then cmd must be in $basic\_cmds \cup send\_cmds$ if $u \neq \mathsf{a}$, and otherwise from $basic\_cmds \cup adv\_local\_cmds \cup adv\_send\_cmds$. Each basic command represents one cryptographic operation; arbitrary "terms" similar to the Dolev-Yao model can be built up or decomposed by a sequence of commands. The sets of commands can be enlarged by further extensions of the library. The permitted argument numbers j and exact domains are defined in Section 4.5. Membership in all domains is easily verifiable.

The inputs at a port $\mathsf{net\_id}_{u,v,x}?$ come from $\mathsf{TH}_{\mathcal{H}}$ itself; each one is simply a message index l^{ind}.

[6]For simulatability, we do not need $\mathsf{TH}_{\mathcal{H}}$ to be polynomial-time; we only need the counters and bounds for the user ports $\mathsf{in}_u?$.

4.5 Inputs and their Evaluation

We now define the precise inputs and how $\mathsf{TH}_\mathcal{H}$ evaluates them based on its abstract state. The main ideas were already explained in Section 2.2. The following more specific design decisions were made, mainly to simplify the system:

- There are commands to store real-world messages in the library and to retrieve them by a handle. Thus other commands can assume that everything is addressed by handles.
- We only allow lists to be signed and transferred, because the list-operation is a convenient place to concentrate all verifications that no secret items are put into messages.
- $\mathsf{TH}_\mathcal{H}$ always offers all types of channels for all participants. Otherwise we would need many different trusted hosts. Many real protocols using cryptography should only use secret and authentic channels for key distribution at the beginning. As the channel type is part of the send-command name, syntactic checks can ensure that a protocol designed with the ideal crypto-library fulfills such requirements.

4.5.1 General Conventions

Upon each input at a port $\mathsf{p}?$, the trusted host $\mathsf{TH}_\mathcal{H}$ first increments the counter $steps_{\mathsf{p}?}$. The machine model in [34] is such that only non-empty inputs count, as only for those a Turing step is needed in the Turing machine realization.

Furthermore, the machine model contains length functions which allow to bound how many bits of input are accepted at each port, depending on the current state. The value 0 for port $\mathsf{p}?$ means that no input is accepted (again without a Turing step). Once a counter $steps_{\mathsf{p}?}$ has reached its bound $bound_{\mathsf{p}?}$, the length function for this port is always zero.

Otherwise, the length functions are determined by the domain specified for each input in the part "for ..." after the parameter list below. The domain of each parameter with the superscript hnd is $\mathcal{HNDS}$, and the handle must be assigned at the time of execution, i.e., $\leq curhnd_u$ in the given state. An input is *well-formed* if it is in the correct domain (at the time of execution). If an input is not well-formed, which is easily verifiable, $\mathsf{TH}_\mathcal{H}$ aborts the transition without further action. (But the counter $steps_{\mathsf{p}?}$ has been incremented.) The overall length function for each port $\mathsf{p}?$ in each state is the maximum of the possible lengths of well-formed inputs at that port in that state; it can easily be computed.

The localized version $\mathsf{TH}_\mathcal{H}^{\mathsf{loc}}$ accompanies every output at a port $\mathsf{out}_u!$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ by an output 1 at $\mathsf{out}_u{}^{\triangleleft}!$, i.e., it schedules that output immediately, the stand-alone version $\mathsf{TH}_\mathcal{H}^{\mathsf{stan}}$ only for $u = \mathsf{a}$.

From now on, we only need to define well-formed inputs further.

4.5.2 Basic Commands

In this section we define how $\mathsf{TH}_\mathcal{H}$ evaluates a command from *basic_cmds*, entered at a port $\mathsf{in}_u?$ with $u \in \mathcal{H} \cup \{\mathsf{a}\}$. These commands are *local*, i.e., they produce a result at port $\mathsf{out}_u!$ and possibly update the database D, but do not produce outputs at other ports. They also do not touch handles for participants $v \neq u$.

Type and length queries

- *Type query:* $t \leftarrow \mathsf{get\_type}(x^{\mathsf{hnd}})$.

Set $t := D[hnd_u = x^{\mathsf{hnd}}].type$.[7]

- *Length query:* $l \leftarrow \mathsf{get\_len}(x^{\mathsf{hnd}})$.

  Set $l := D[hnd_u = x^{\mathsf{hnd}}].len$.

Storing and retrieving data $\mathsf{TH}_{\mathcal{H}}$ stores a string and gives u a handle to it, respectively retrieves a previously stored string.

- *Storing:* $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$, for $m \in \mathcal{BITSTR}$ with $|m| \leq \mathsf{max\_len}(k)$.

  If $i := D[type = \mathsf{data} \wedge arg = (m)].ind \neq \downarrow$ then return $m^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(i)$.[8] Otherwise if $\mathsf{data\_len}^*(|m|) > \mathsf{max\_len}(k)$ return $\downarrow$. Else set $m^{\mathsf{hnd}} := curhnd_u{+}{+}$ and

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{data}, arg := (m), hnd_u := m^{\mathsf{hnd}},$$
$$len := \mathsf{data\_len}^*(|m|)).$$

- *Retrieval:* $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  $m := D[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].arg[1]$.[9]

Lists Entries that can be sent to other players, like data, ciphertexts, and public keys, can be combined into lists. Lists cannot include secret keys because no information about those must be given away. The following commands create a list and determine the i-th element in a list.

- *Generate a list:* $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, \ldots, x_j^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.

  Let $x_i := D[hnd_u = x_i^{\mathsf{hnd}}].ind$ for $i = 1, \ldots, j$. If any $D[x_i].type \in \{\mathsf{sks}, \mathsf{ske}\}$, set $l^{\mathsf{hnd}} := \downarrow$.

  If $l := D[type = \mathsf{list} \wedge arg = (x_1, \ldots, x_j)].ind \neq \downarrow$, then return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$. Otherwise, set $length := \mathsf{list\_len}^*(D[x_1].len, \ldots, D[x_j].len)$ and return $\downarrow$ if $length > \mathsf{max\_len}(k)$. Else set $l^{\mathsf{hnd}} := curhnd_u{+}{+}$ and

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{list}, arg := (x_1, \ldots, x_j), hnd_u := l^{\mathsf{hnd}},$$
$$len := length).$$

- *i-th projection:* $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $1 \leq i \leq \mathsf{max\_len}(k)$.

  If $D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].arg = (x_1, \ldots, x_j)$ with $j \geq i$, then $x^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(x_i)$, otherwise $x^{\mathsf{hnd}} := \downarrow$.

Nonces In reality nonces are randomly chosen words, used, e.g., to verify freshness of messages. $\mathsf{TH}_{\mathcal{H}}$ just creates a new entry.

- *Generation:* $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Set $n^{\mathsf{hnd}} := curhnd_u{+}{+}$ and

$$D :\Leftarrow (ind := size{+}{+}, type := \mathsf{nonce}, arg := (), hnd_u := n^{\mathsf{hnd}},$$
$$len := \mathsf{nonce\_len}^*(k)).$$

[7]Note that u cannot get the type of entries he does not know because $D^*[hnd_u = x^{\mathsf{hnd}}] = \downarrow$ implies $t = \downarrow$.

[8]Hence if the same string m is stored twice, $\mathsf{TH}_{\mathcal{H}}$ reuses the first result. In general $\mathsf{TH}_{\mathcal{H}}$ ensures that two identical pieces of data get the same index. However, encryption or signing always create new entries, because these operations are indeterministic.

[9]This implies that m^{hnd} was created by a store command, as no other command creates entries with $type = \mathsf{data}$. Thus only explicitly stored data can be retrieved and not, e.g., keys or ciphertexts.

Signatures This comprises operations for key generation, signature generation and verification, and retrieval of signature components.

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_sig\_keypair}()$.

  Set $sk^{\mathsf{hnd}} := curhnd_u$++; $pk^{\mathsf{hnd}} := curhnd_u$++ and

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{sks}, arg := (0), hnd_u := sk^{\mathsf{hnd}}, len := 0);$$

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{pks}, arg := (), hnd_u := pk^{\mathsf{hnd}}, len := \mathsf{pks\_len}^*(k)).$$

  The argument of the secret-key entry is a counter initialized with 0.

- *Signature generation:* $s^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $sk := D[hnd_u = sk^{\mathsf{hnd}} \wedge type = \mathsf{sks}].ind$ and $l := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If either of these is $\downarrow$ or if $length := \mathsf{sig\_len}^*(k, D[l].len) > \mathsf{max\_len}(k)$, return $\downarrow$. Also return $\downarrow$ if $D[sk].arg[1] \geq \mathsf{max\_skc}(k)$ and $u \neq \mathsf{a}$. Otherwise, set $s^{\mathsf{hnd}} := curhnd_u$++, $pk := sk + 1$ (recall that key pairs get successive indices), $c := D[sk].arg[1]$++, and

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{sig}, arg := (pk, l, c), hnd_u := s^{\mathsf{hnd}}, len := length).$$[10]

- *Signature verification:* $v \leftarrow \mathsf{verify}(s^{\mathsf{hnd}}, pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $s := D[hnd_u = s^{\mathsf{hnd}} \wedge type = \mathsf{sig}].ind$. If $s = \downarrow$ then return $\downarrow$. Otherwise, let $(pk, l, c) := D[s].arg$. If $D[pk].hnd_u \neq pk^{\mathsf{hnd}}$ or $D[l].hnd_u \neq l^{\mathsf{hnd}}$, then $v := \mathsf{false}$, else $v := \mathsf{true}$.[11]

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_sig}(s^{\mathsf{hnd}})$.

  Let $pk := D[hnd_u = s^{\mathsf{hnd}} \wedge type = \mathsf{sig}].arg[1]$ and return $pk^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(pk)$.

- *Message retrieval:* $l^{\mathsf{hnd}} \leftarrow \mathsf{msg\_of\_sig}(s^{\mathsf{hnd}})$.

  Let $l := D[hnd_u = s^{\mathsf{hnd}} \wedge type = \mathsf{sig}].arg[2]$ and return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.[12,13]

Public-key encryption This comprises operations for key generation, encryption, decryption, and retrieval of the public key from a ciphertext.

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$.

  Sets $sk^{\mathsf{hnd}} := curhnd_u$++, $pk^{\mathsf{hnd}} := curhnd_u$++, and

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{ske}, arg := (), hnd_u := sk^{\mathsf{hnd}}, len := 0);$$

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{pke}, arg := (), hnd_u := pk^{\mathsf{hnd}}, len := \mathsf{pke\_len}^*(k)).$$

- *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $pk := D[hnd_u = pk^{\mathsf{hnd}} \wedge type = \mathsf{pke}].ind$ and $l := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$ and $length := \mathsf{enc\_len}^*(k, D[l].len)$. If $length > \mathsf{max\_len}(k)$ or $pk = \downarrow$ or $l = \downarrow$, then return $\downarrow$. Else set $c^{\mathsf{hnd}} := curhnd_u$++ and

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{enc}, arg := (pk, l), hnd_u := c^{\mathsf{hnd}}, len := length).$$[14]

[10]This type also exists with $c = \mathsf{false}$ due to the command adv_transform_sig.

[11]Given the next two commands, signature verification is redundant. This works because for received real alleged signatures, the type sig and the arguments are only assigned after cryptographic verification; otherwise the type becomes garbage.

[12]This command implies that real signatures must contain the message, which may seem inefficient. However, the simulator needs this to translate signatures from the adversary into abstract ones. Thus we also offer message retrieval to honest users so that they need not send the message separately.

[13]A similar operation for retrieving the counter is only for the adversary because not all signature schemes allow it.

[14]This type also has a second format due to the command adv_invalid_ciph.

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  Let $sk := D[hnd_u = sk^{\mathsf{hnd}} \wedge type = \mathsf{ske}].ind$ and $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$. Return $\downarrow$ if $c = \downarrow$ or $sk = \downarrow$ or $pk := D[c].arg[1] \neq sk + 1$ or $l := D[c].arg[2] = \downarrow$ (as a result of the command $\mathsf{adv\_invalid\_ciph}$). Else return $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_enc}(c^{\mathsf{hnd}})$.

  Let $c := D[hnd_u = c^{\mathsf{hnd}} \wedge type = \mathsf{enc}].ind$. If $c = \downarrow$, return $\downarrow$. Otherwise, let $pk := D[c].arg[1]$ and return $pk^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(pk)$.

4.5.3 Local Adversary Commands

The following commands are also local, but, as defined in Section 4.4, only well-formed at the port $\mathsf{in_a}?$. Hence they are not available as commands for the honest users in the real system. They model tolerable imperfections of the real system, i.e., actions that may be possible in real systems but that are not required.

First, an adversary may create invalid entries of a certain length; they obtain the type $\mathsf{garbage}$. Secondly, invalid ciphertexts are a special case because participants not knowing the secret key can reasonably ask for their type and query their public key, hence they cannot be of type $\mathsf{garbage}$. Thirdly, the security definition of signature schemes does not exclude that the adversary transforms signatures by honest participants into other valid signatures on the same message with the same public key.

Finally, we allow the adversary to retrieve all information that we do not explicitly require to be hidden. We write this as a command $\mathsf{adv\_parse}$. It returns the type and usually all the abstract arguments of a value (with indices replaced by handles), except that for ciphertexts where the adversary does not know the secret key it only returns the length of the cleartext instead of the cleartext itself. Most of these arguments could also be queried with basic commands, but not the signature counter and the cleartext length.

- *Invalid entry:* $y^{\mathsf{hnd}} \leftarrow \mathsf{adv\_garbage}(l)$, for $l \in \mathbb{N}$ with $l \leq \mathsf{max\_len}(k)$.

  Set $y^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$ and[15]

  $$D :\Leftarrow (ind := size\text{++}, type := \mathsf{garbage}, arg := (), hnd_{\mathsf{a}} := y^{\mathsf{hnd}}, len := l).$$

- *Signature transformation:* $t^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_sig}(s^{\mathsf{hnd}})$.

  Let $s := D[hnd_{\mathsf{a}} = s^{\mathsf{hnd}} \wedge type = \mathsf{sig}].ind$. If $s = \downarrow$ then return $\downarrow$. Otherwise let $(pk, l, c) := D[s].arg$. Set $t^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$ and

  $$\begin{aligned} D \;:\Leftarrow\; & (ind := size\text{++}, type := \mathsf{sig}, arg := (pk, l, \mathsf{false}), hnd_{\mathsf{a}} := t^{\mathsf{hnd}}, \\ & len := D[s].len). \end{aligned}$$

- *Invalid ciphertext of length l:* $c^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_ciph}(pk^{\mathsf{hnd}}, l)$, for $1 \leq l \leq \mathsf{max\_len}(k)$.

  Let $pk := D[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}} \wedge type = \mathsf{pke}].ind$. If $pk = \downarrow$, then return $\downarrow$. Otherwise set $c^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$ and

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

[15]This does not mean that the same garbage is always treated differently, only that the simulation has to ensure that this command is only used for new garbage.

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

  Let $m := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].ind$ and $type := D[m].type$. In most cases, set $arg := \mathsf{ind2hnd}^*_{\mathsf{a}}(D[m].arg)$. (Recall that this only transforms arguments in $\mathcal{INDS}$.) The only exception is for $type = \mathsf{enc}$ and $D[m].arg$ of the form (pk, l) (a valid ciphertext) and $D[pk-1].hnd_{\mathsf{a}} = \downarrow$ (the adversary does not know the secret key); then $arg := (\mathsf{ind2hnd_a}(pk), D[l].len)$.

4.5.4 Send Commands

We now define the send commands in detail. Essentially, messages from one honest user to another (on a secure or authentic channel) are put on an abstract secure channel for scheduling, while messages to or from the adversary (either by their addresses, or because the channel is not private) are output immediately.

- $\mathsf{send\_}x(v, l^{\mathsf{hnd}})$, for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$ and $v \in \{1, \ldots, n\}$.

  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 depending on the channel type (where both cases may occur for one message):

  – If $(u, v, x) \in ch\_honest$, output l^{ind} at $\mathsf{net\_id}_{u,v,x}!$.
  – If $(u, v, x) \in ch\_to\_adv$, output $(u, v, x, \mathsf{ind2hnd_a}(l^{\mathsf{ind}}))$ at $\mathsf{out_a}!$.

- $\mathsf{adv\_send\_}x(u, v, l^{\mathsf{hnd}})$, for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$, $u \in \{1, \ldots, n\}$ and $v \in \mathcal{H}$ at port $\mathsf{in_a}?$.

  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 $l^{\mathsf{ind}} \neq \downarrow$ and $(u, v, x) \in ch\_from\_adv$, output $(u, v, x, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at $\mathsf{out}_v!$.

4.5.5 Inputs from Secure Channels

- *Input from secure channel:* On input l^{ind} at a port $\mathsf{net\_id}_{u,v,x}?$, for $l^{\mathsf{ind}} \in \mathcal{INDS}$ with $l^{\mathsf{ind}} \leq size$.

  $\mathsf{TH}_{\mathcal{H}}$ outputs $(u, x, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at $\mathsf{out}_v!$.

4.6 Properties of the Ideal System

The following properties of the ideal system will be useful as invariants in the security proof. For every entry x in D, let $\mathsf{owners}(x) := \{u \in \mathcal{H} \cup \{\mathsf{a}\} \mid x.hnd_u \neq \downarrow\}$.

Lemma 4.1 The ideal systems $Sys^{\mathsf{cry,id},x}_{n,L}$ for $x \in \{\mathsf{stan}, \mathsf{loc}\}$ have the following properties:

1. *Index and handle uniqueness*: The argument ind of D is a key attribute. Further, D always fulfills $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 1$ for each $i^{\mathsf{hnd}} \leq curhnd_u$, and $|\sigma_{hnd_u = i^{\mathsf{hnd}}}(D)| = 0$ for each $i^{\mathsf{hnd}} > curhnd_u$.

2. *Well-defined terms*: If an entry $x = D[i]$ has an index argument, i.e., $a := x.arg[j] \in \mathcal{INDS}$ for some $j \in \mathbb{N}$, then $a < i$. (Thus following the arguments of an entry backwards recursively yields a finite, Dolev-Yao-like term.) We call the entry $D[a]$ a direct component of x, and recursively define the set of all components of x in the obvious way.

3. *Message correctness*: At all times and for all $(u, v, x) \in ch\_honest$, each message $l^{\mathsf{ind}} := net\_id_{u,v,x}[i]$ with $l^{\mathsf{ind}} \neq \downarrow$ has $D[l^{\mathsf{ind}}].type = \mathsf{list}$. Here, $net\_id_{u,v,x}[i]$ denotes the i-th element in the buffer queue of the connection.

4. *Length bounds:* At all times, $x.len \leq \mathsf{max\_len}(k)$ for all $x \in D$.

5. *Correct key pairs:* Key entries only exist in pairs, i.e., $D[i].type = \mathsf{sks} \iff D[i+1].type = \mathsf{pks}$ and $D[i].type = \mathsf{ske} \iff D[i+1].type = \mathsf{pke}$ at all times and for all $i \in \mathbb{N}_0$. If the conditions are true, we further have $D[i+1].hnd_u = D[i].hnd_u + 1$ for $u := \mathsf{owner}(D[i])$.

6. *Key secrecy*: If $D[i].type \in \{\mathsf{sks}, \mathsf{ske}\}$, then $D[i]$ is not a component of $D[j]$ for any $j \neq i$. Further, $|\mathsf{owners}(D[i])| = 1$. We write $\mathsf{owner}(D[i])$ for the element of $\mathsf{owners}(D[i])$ in this case. It cannot change later.

□

Proof. Part 1 was already shown in Sections 4.3.2 and 4.3.3; it is easy to see that all commands respect the conventions introduced there.

Parts 2, 3, 4, and 5 can easily be seen by inspection of the commands, and for Part 4 by the precondition on the lengths of nonces and keys.

For Part 6, inspection easily shows that no type of entry has a direct component of a type in $\{\mathsf{sks}, \mathsf{ske}\}$, and thus also no component. Handles may be added to existing entries upon the following inputs: Data storing and list generation, but only to data and lists; list projection, decryption, retrieval from signatures and ciphertexts, and adv_parse, but only to direct components of the given entry and thus not to secret keys; send_x and adv_send_x, but only to lists; and network inputs, but only to lists by Part 3. ■

Moreover, the following properties will prove useful.

Lemma 4.2 The ideal systems $Sys_{n,L}^{\mathsf{cry,id,stan}}$ and $Sys_{n,L}^{\mathsf{cry,id,loc}}$ have the following properties:

1. The systems are polynomial-time.

2. No ideal network input is rejected because a counter $steps_{\mathsf{net\_id}_{u,v,x}?}$ reached its bound.

3. The only modifications to existing entries x in D are assignments to previously undefined attributes $x.hnd_u$, and (via the command sign) counter updates in entries of type sks.

4. Whenever $\mathsf{TH}_{\mathcal{H}}$ assigns a handle $x.hnd_u$, it outputs it at port $\mathsf{out}_u!$ in the same transition. Further, every new entry has exactly one handle.

□

Proof. For Part 1, we have to show that $\mathsf{TH}_{\mathcal{H}}$ only accepts a polynomial number of polynomial-length inputs; the action in each command is then clearly polynomial-time by the assumptions about the functions in L. For the input number, this holds because there is a counter $steps_{\mathsf{p}?}$ for inputs at each port $\mathsf{p}?$, each with a bound $bound_{\mathsf{p}?}$ polynomial in k. Hence the dynamic bounds $curhnd_u$ on handle parameters and $size$ on ideal network inputs are bounded by a statically fixed polynomial. All other input parameters have static polynomial domain restrictions. Hence the length bounds on inputs (recall Section 4.5.1) are indeed polynomial in k.

Part 2 holds because messages only get into a buffer $net\_id_{u,v,x}$ by a command send_x input at $\mathsf{in}_u?$, one per command. There are at most $\mathsf{max\_in}(k)$ such inputs, and $\mathsf{max\_in}(k)$ messages are accepted at $\mathsf{net\_id}_{u,v,x}?$ if scheduled.

Parts 3 and 4 can easily be seen by inspection of the commands. ■

5 Real System

In a real system, the commands are implemented by real cryptographic algorithms, and messages are actually sent between machines. We first describe the underlying algorithms; then we define the machines of the real system in an order similar to the ideal system.

5.1 Cryptographic Operations

The real system uses digital signatures and public-key encryption. The ranges of all algorithms are $\{0,1\}^+ \cup \{\downarrow\}$.

5.1.1 Digital Signatures

We denote a digital signature scheme by a tuple $\mathcal{S} = (\mathsf{gen_S}, \mathsf{sign}, \mathsf{test}, \mathsf{pks\_len}, \mathsf{sig\_len})$ of polynomial-time algorithms. We write

$$(sk, pk) \leftarrow \mathsf{gen_S}(1^k, 1^s)$$

for the generation of a secret signing key and a public test key based on a security parameter, $k \in \mathbb{N}$, and the desired maximum number of signatures, $s \in \mathbb{N}$. The length of pk is $\mathsf{pks\_len}(k, s) > 0$. By

$$sig \leftarrow \mathsf{sign}_{sk,sc}(m)$$

we denote the (probabilistic) signing of a message $m \in \{0,1\}^+$, where $sc \in \{1, \ldots, s\}$ is a counter value.[16] Verification

$$b := \mathsf{test}_{pk}(sig, m)$$

is deterministic and returns true (then we say that the signature is valid) or false. Correctly generated signatures for correct key pairs must always be valid. The length of sig is $\mathsf{sig\_len}(k, s, |m|) > 0$.[17] This is also true for every sig' with $\mathsf{test}_{pk}(sig', m) = \mathsf{true}$ for a value $pk \in \{0,1\}^{\mathsf{pks\_len}(k,s)}$. The functions $\mathsf{pks\_len}$, $\mathsf{sig\_len}$ must be bounded by multivariate polynomials.

The accepted security definition for general-purpose signing is security against existential forgery under adaptive chosen-message attacks [21]. We only use our notation for interacting machines.

Definition 5.1 *(Signature Security) Given a signature scheme and a function s over $\mathbb{N}$, the* signer machine Sig_s *is defined as follows: It has one input and one output port, variables sk, pk initialized with $\downarrow$ and a counter sc initialized with 0, and the following transition rules:*

- *First generate a key pair, $(sk, pk) \leftarrow \mathsf{gen_S}(1^k, 1^{s(k)})$, and output pk.*
- *On input (sign, m_j), and if $sc < s(k)$, set sc++ and return $sig_j \leftarrow \mathsf{sign}_{sk,sc}(m_j)$.*

The signature scheme is called existentially unforgeable under adaptive chosen-message attack *if for every efficiently computable, polynomially bounded s and every probabilistic polynomial-time machine $\mathsf{A_{sig}}$ that interacts with Sig_s and outputs a pair (m, sig), the probability that $\mathsf{test}_{pk}(sig, m) = \mathsf{true}$ and m was not signed by Sig_s during the interaction (i.e., a forged signature), is negligible (in k).* $\diamond$

[16] The most efficient implementations of typical signature schemes needing counters also store a path in a tree. Functionally, this or any other function of random values chosen during earlier applications of sign_{sk} is equivalent to just a counter, because all random values needed can be seen as part of sk and chosen initially.

[17] For real signature schemes one often gets only upper bounds on these lengths. By appropriate padding one can easily ensure that these bounds are met exactly.

5.1.2 Encryption

We denote a public-key encryption scheme by a tuple $\mathcal{E} = (\mathsf{gen}_\mathsf{E}, \mathsf{E}, \mathsf{D}, \mathsf{pke\_len}, \mathsf{enc\_len})$ of polynomial-time algorithms. We assume that messages of arbitrary length can be encrypted and that secrecy holds within all subsets of messages of the same length. We write

$$(sk, pk) \leftarrow \mathsf{gen}_\mathsf{E}(1^k)$$

for the generation of a secret decryption key and a public encryption key. The length of pk is $\mathsf{pke\_len}(k) > 0$. We denote the (probabilistic) encryption of a message $m \in \{0,1\}^+$ by

$$c \leftarrow \mathsf{E}_{pk}(m),$$

and deterministic decryption by

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

The result m may be $\downarrow$ for wrong ciphertexts. The length of c is $\mathsf{enc\_len}(k, |m|) \geq |m| > 0$ for every value $pk \in \{0,1\}^{\mathsf{pke\_len}(k)}$, and also for every c' with $m := \mathsf{D}_{sk}(c') \neq \downarrow$ for a correctly generated secret key. The functions $\mathsf{pke\_len}$, $\mathsf{enc\_len}$ must be bounded by multivariate polynomials.

The following security definition means that any two equal-length messages are indistinguishable even in adaptive chosen-ciphertext attacks. Indistinguishability was introduced in [20], chosen-ciphertext security in [35] and formalized as "IND-CCA2" in [8]. It is the accepted definition for general-purpose encryption. An efficient encryption system secure in this sense is [13].

Definition 5.2 *(Encryption Security) Given an encryption scheme, the* decryptor machine Dec *is defined as follows: It has one input and one output port, variables sk, pk, c initialized with $\downarrow$, and the following transition rules:*

- *First set $(sk, pk) \leftarrow \mathsf{gen}_\mathsf{E}(1^k)$ and output pk.*
- *On input (enc, m_0, m_1) (intuitively a pair of messages an adversary hopes to be able to distinguish), and if $|m_0| = |m_1|$ and $c = \downarrow$, set $b \xleftarrow{\mathcal{R}} \{0,1\}$ and store and output the encryption $c \leftarrow \mathsf{E}_{pk}(m_b)$.*
- *On input (dec, c_j) and if $c_j \neq c$, return $m := \mathsf{D}_{sk}(c)$.*

The encryption scheme is called indistinguishable under adaptive chosen-ciphertext attack *if for every probabilistic polynomial-time machine A_enc that interacts with* Dec *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.* $\diamond$

5.2 Structures

Given a signature scheme $\mathcal{S}$ and an encryption scheme $\mathcal{E}$, we now define a real crypto-library, again in a stand-alone and a localized version $Sys_{n,\mathcal{S},\mathcal{E},L'}^{\mathsf{cry,real,stan}}$ and $Sys_{n,\mathcal{S},\mathcal{E},L'}^{\mathsf{cry,real,loc}}$. The parameter n denotes the number of participants, and L' is a tuple of parameters discussed in Section 5.3.

The systems are standard cryptographic systems as defined in [34] and sketched in Section 2.1.

- The intended structure has n machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$.
- The intended specified ports are defined via

$$S^{*,\mathsf{stan}^C} := \{\mathsf{in}_u!, \mathsf{out}_u? \mid u \in \{1, \ldots n\}\};$$

$$S^{*,\mathsf{loc}^C} := \{\mathsf{in}_u!, \mathsf{out}_u?, \mathsf{in}_u{}^{\triangleleft}! \mid u \in \{1, \ldots n\}\}.$$

- Correspondingly, each M_u has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$, and in the localized version also $\mathsf{out}_u{}^{\triangleleft}!$
- Each M_u has three connections to each M_v with $v \in \{1, \ldots n\}$, called $net_{u,v,\mathsf{s}}$, $net_{u,v,\mathsf{a}}$ and $net_{u,v,\mathsf{i}}$. They are called *network connections* and the corresponding ports *network ports*. All the network connections are scheduled by the adversary.
- The access structure $\mathcal{ACC}$ consists of all subsets $\mathcal{H}$ of $\{1, \ldots, n\}$.
- The channel model χ maps each connection $net_{u,v,x}$ to x.

The channel model means that in an actual structure, an honest intended recipient gets all messages output at ports of type s (secret) and a (authentic) and the adversary gets all messages output at ports of type a and i (insecure). Furthermore, all messages received at an input port of type i come from the adversary. This is shown in Figure 3. To ensure globally unique port names, some ports get a superscript a according to the rules in [34]. In the following, the machine is always clear from the context; thus we usually omit these superscripts.

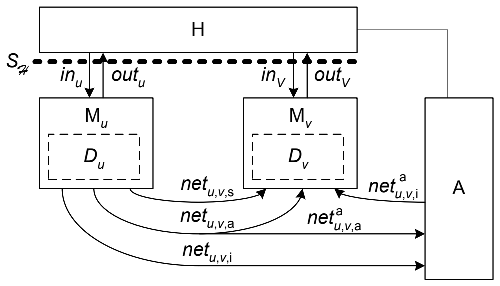

Figure 3: Connections from a correct machine to another in the real system. User in- and outputs are scheduled as in the ideal system, network connections by the adversary.

5.3 Lengths and Bounds

The parameter L' of the real system consists of functions $\mathsf{max\_len}(k)$, $\mathsf{max\_skc}(k)$, and $\mathsf{max\_in}(k)$ as in the ideal system and a function $\mathsf{nonce\_len}(k)$ of the same type, where $2^{-\mathsf{nonce\_len}(k)}$ must be negligible.

For use in preconditions and lemmas, we define the resulting lengths of data items in the real system, i.e., the length functions of the corresponding ideal system, as a function $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$. We abbreviate

- $\mathsf{pks\_len}'(k) := \mathsf{pks\_len}(k, \mathsf{max\_skc}(k))$;
- $\mathsf{sig\_len}'(k, l) := \mathsf{sig\_len}(k, \mathsf{max\_skc}(k), \mathsf{list\_len}(\mathsf{nonce\_len}(k), l))$; and
- $\mathsf{enc\_len}'(k, l) := \mathsf{enc\_len}(k, \mathsf{list\_len}(\mathsf{nonce\_len}(k), l))$.

Now L consists of the following functions:

- $\mathsf{data\_len}^*(l) := \mathsf{list\_len}(|\mathsf{data}|, l)$;
- $\mathsf{list\_len}^*(l_1, \ldots, l_j) := \mathsf{list\_len}(|\mathsf{list}|, l_1, \ldots, l_j)$;
- $\mathsf{nonce\_len}^*(k) := \mathsf{list\_len}(|\mathsf{nonce}|, \mathsf{nonce\_len}(k))$;
- $\mathsf{pks\_len}^*(k) := \mathsf{list\_len}(|\mathsf{pks}|, \mathsf{pks\_len}'(k))$;

- $\mathsf{sig\_len}^*(k,l) := \mathsf{list\_len}(|\mathsf{sig}|, \mathsf{pks\_len}'(k), \mathsf{nonce\_len}(k), l, \mathsf{sig\_len}'(k,l))$;
- $\mathsf{pke\_len}^*(k) := \mathsf{list\_len}(|\mathsf{pke}|, \mathsf{pke\_len}(k))$;
- $\mathsf{enc\_len}^*(k,l) := \mathsf{list\_len}(|\mathsf{enc}|, \mathsf{pke\_len}(k), \mathsf{enc\_len}'(k,l), \mathsf{nonce\_len}(k))$;

and $\mathsf{max\_len}$, $\mathsf{max\_skc}$ and $\mathsf{max\_in}$ are taken unchanged from L'.

We require that $\mathsf{nonce\_len}^*(k)$, $\mathsf{pks\_len}^*(k)$, and $\mathsf{pke\_len}^*(k)$ are upper bounded by $\mathsf{max\_len}(k)$.

Lemma 5.1 For a correct tuple L' of parameters for the real crypto-library and correct signature and encryption systems, the algorithm $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$ yields a correct tuple L of parameters for the ideal crypto-library. □

Proof. We have to verify the conditions from Section 4.2. Clearly the range of all functions $\mathit{type}\_\mathsf{len}^*$ is indeed $\mathbb{N}$. Next, $\mathit{type}\_\mathsf{len}^*(k) < \mathsf{max\_len}(k)$ for $\mathit{type} \in \{\mathsf{nonce}, \mathsf{pks}, \mathsf{pke}\}$ and $\mathsf{max\_skc}(k) < \mathsf{max\_in}(k)$ holds by direct preconditions. Finally, one easily sees that all the functions are bounded by a multivariate polynomial and efficiently computable because all the basic functions in the definitions are. ■

5.4 States of a Machine

The state of each machine M_u consists of a database D_u and variables curhnd_u and $\mathit{steps}_{\mathsf{p}?}$ for each input port $\mathsf{p}?$.

5.4.1 Database D_u

The main data structure of M_u is a database D_u that contains implementation-specific data such as ciphertexts and signatures produced during a system run. Its primary key attribute corresponds to the handles for u in the ideal system. Each entry x in D_u has attributes

$$(\mathit{hnd}_u, \mathit{word}, \mathit{type}, \mathit{add\_arg}).$$

- $x.\mathit{hnd}_u \in \mathcal{HNDS}$ consecutively numbers all entries in D_u. We use it as a primary key attribute, i.e., we will write $D_u[i^{\mathsf{hnd}}]$ for $D_u[\mathit{hnd}_u = i^{\mathsf{hnd}}]$.
- $x.\mathit{word} \in \{0,1\}^+$ is the real representation of x.
- $x.\mathit{type} \in \mathit{typeset} \cup \{\mathsf{null}\}$ identifies the type of x. The value null denotes that the entry has not yet been parsed. Again, the typeset can be extended.
- $x.\mathit{add\_arg}$ is a list of ("additional") arguments. Typically it is (), only for signing keys it contains the counter.

5.4.2 Conventions about Handles

Initially, D_u is empty. M_u maintains a counter curhnd_u of type $\mathcal{HNDS}$ denoting the current number of elements in D_u. New entries x always receive $x.\mathit{hnd}_u := \mathit{curhnd}_u$++, and $x.\mathit{hnd}_u$ is never changed. Thus, hnd_u is indeed a key attribute. By

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

spoken "determine a handle i^{hnd} for ... in D_u", for $i \in \{0,1\}^+$, $\mathit{type} \in \mathit{typeset}$, and $\mathit{add\_arg} \in \mathcal{LIST}$, we abbreviate the following algorithm: If $i^{\mathsf{hnd}} := D_u[\mathit{word} = i \wedge \mathit{type} \notin \{\mathsf{sks}, \mathsf{ske}\}].\mathit{hnd}_u \neq \downarrow$, return i^{hnd}, assigning the input values type and $\mathit{add\_arg}$ to the corresponding attributes of $D_u[i^{\mathsf{hnd}}]$ only if $D_u[i^{\mathsf{hnd}}].\mathit{type}$ was null. Else if $|i| > \mathsf{max\_len}(k)$, return $i^{\mathsf{hnd}} = \downarrow$. Otherwise, set $i^{\mathsf{hnd}} := \mathit{curhnd}_u$++, $D_u :\Leftarrow (i^{\mathsf{hnd}}, i, \mathit{type}, \mathit{add\_arg})$ and return i^{hnd}.

5.4.3 Input Counters

For each of its input ports $\mathsf{p}?$, M_u maintains a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ initialized with 0 for the number of inputs at that port. All corresponding bounds $bound_{\mathsf{p}?}$ are $\mathsf{max\_in}(k)$.

5.5 Inputs and their Evaluation

Now we describe how M_u evaluates individual inputs. Mainly, these are the cryptographic operations with the modifications motivated in Section 2.3. For general unambiguity, not only all cryptographic objects are tagged, but also data and lists.

5.5.1 General Conventions

The general conventions are similar to those in the ideal system. (In fact, the domains for inputs at port $\mathsf{in}_u?$ are the same as in $\mathsf{TH}_{\mathcal{H}}$ in corresponding states, which will be defined later.)

Upon each input at a port $\mathsf{p}?$, the machine M_u first increments the counter $steps_{\mathsf{p}?}$, and once a counter $steps_{\mathsf{p}?}$ has reached its bound $\mathsf{max\_in}(k)$, the length function for this port is always zero.

Otherwise, the length function is determined by the domain specified for each input in the part "for ..." below. The domain of each parameter with subscript hnd is $\mathcal{HNDS}$, and it must be $\leq curhnd_u$ at the time of execution. An input in the correct domain is called *well-formed.* If an input is not well-formed, which is easily verifiable, M_u aborts the transition without further action. (But the counter $steps_{\mathsf{p}?}$ has been incremented.) The overall length function for each port $\mathsf{p}?$ in each state is the maximum of the possible lengths of well-formed inputs at that port in that state; it can easily be computed.

In the localized version, M_u accompanies every output at the port $\mathsf{out}_u!$ by an output 1 at $\mathsf{out}_u{}^{\triangleleft}!$, i.e., it schedules that output immediately.

From now on, we only need to consider well-formed inputs further.

5.5.2 Constructors and One-level Parsing

Constructors For each type, we define the main constructor via an algorithm $\mathsf{make\_}type$, which is purely functional, i.e., it does not use or modify global variables, except that it uses k.[18]

Functional parsing We also define a functional algorithm parse, written

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

for $m \in \{0,1\}^+$ and with outputs $type \in typeset$ and a list arg with elements in Σ^*.

On input m, it tests if m is of the form $(type, m_1, \ldots, m_j)$ with $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ and $j \geq 0$. If this fails, it returns $(\mathsf{garbage}, ())$. Otherwise it makes a call $arg \leftarrow \mathsf{parse\_}type(m)$ to an algorithm which returns $\downarrow$ if m is not really of this type and otherwise computes the arguments. These algorithms are described after the corresponding constructors in Section 5.5.3. If $\mathsf{parse\_}type$ returns $arg = \downarrow$, parse again outputs $(\mathsf{garbage}, ())$.

Parsing in D_u By

$$\text{“parse } m^{\mathsf{hnd}}\text{”}$$

[18]This is mainly useful for the security proof. An implementation may be inline.

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

"parse m^{hnd} if necessary"

we mean the same except that M_u does nothing if $D_u[m^{\mathsf{hnd}}].type \neq \mathsf{null}$.[19]

5.5.3 Basic Commands and parse_type

We now define how M_u evaluates a command from *basic_cmds*, entered at the port $\mathsf{in}_u?$. Basic commands are again *local*, i.e., they produce a result at $\mathsf{out}_u!$ and possibly update the database D_u, but do not produce outputs at other ports.

Type and length queries

- *Type query:* $t \leftarrow \mathsf{get\_type}(x^{\mathsf{hnd}})$.

  Parse x^{hnd} if necessary. Then let $t := D_u[x^{\mathsf{hnd}}].type$.

- *Length query:* $l \leftarrow \mathsf{get\_len}(x^{\mathsf{hnd}})$.

  Parse x^{hnd} if necessary. If $D_u[x^{\mathsf{hnd}}].type \not\in \{\mathsf{sks}, \mathsf{ske}\}$ then let $l := |D_u[x^{\mathsf{hnd}}].word|$, otherwise let $l := 0$.

Storing and retrieving data

- *Constructor:* $d \leftarrow \mathsf{make\_data}(m)$, for $m \in \{0,1\}^*$.

  Let $d := (\mathsf{data}, m)$.

- *Storing:* $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$, for $m \in \{0,1\}^*$ with $|m| \leq \mathsf{max\_len}(k)$.

  Let $d \leftarrow \mathsf{make\_data}(m)$ and $(d^{\mathsf{hnd}}, D_u) :\leftarrow (d, \mathsf{data}, ())$.

- *Parsing:* $arg \leftarrow \mathsf{parse\_data}(m)$.

  If m is of the form (data, m') with $m' \in \{0,1\}^*$, return (m'), else $\downarrow$.

- *Retrieval:* $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  Parse m^{hnd} if necessary. Return $\mathsf{parse\_data}(D_u[hnd_u = m^{\mathsf{hnd}} \wedge type = \mathsf{data}].word)[1]$.

Lists

- *Constructor:* $l \leftarrow \mathsf{make\_list}(x_1, \ldots, x_j)$, for $j \in \mathbb{N}_0$ and $x_i \in \{0,1\}^+$ for $i := 1, \ldots, j$.

  Let $l := (\mathsf{list}, x_1, \ldots, x_j)$.

- *Generate a list:* $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, \ldots, x_j^{\mathsf{hnd}})$, for $0 \leq j \leq \mathsf{max\_len}(k)$.

  If there is an i with $D_u[x_i^{\mathsf{hnd}}].type \in \{\mathsf{sks}, \mathsf{ske}\}$, return $\downarrow$.[20] Else set $l := \mathsf{make\_list}(D_u[x_1^{\mathsf{hnd}}].word, \ldots, D_u[x_j^{\mathsf{hnd}}].word)$ and $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

- *Parsing:* $arg \leftarrow \mathsf{parse\_list}(l)$.

  If l is of the form $(\mathsf{list}, x_1, \ldots, x_j)$ with $j \in \mathbb{N}_0$ and $x_i \in \{0,1\}^+$ for $i := 1, \ldots, j$, return $(x_1, \ldots, x_j)$, else $\downarrow$. In the first case, we call l a *tagged list*.

- *i-th projection:* $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $1 \leq i \leq \mathsf{max\_len}(k)$.

  Parse l^{hnd}, yielding arg. If $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, return $\downarrow$. Otherwise let $x := arg[i]$. If $x = \downarrow$, return $\downarrow$, else let $(x^{\mathsf{hnd}}, D_u) :\leftarrow (x, \mathsf{null}, ())$.

[19] This notation is mainly used for reasons of readability. Both algorithms can be formalized as $arg \leftarrow \mathsf{parse}_{D_u}(m^{\mathsf{hnd}})$ and $\mathsf{parse\_cond}_{D_u}(m^{\mathsf{hnd}})$ (with side effects).

[20] We need not parse the inputs because parsing never gives types from $\{\mathsf{sks}, \mathsf{ske}\}$, and garbage in lists is allowed.

Nonces

- *Constructor:* $n \leftarrow \mathsf{make\_nonce}()$.

  Let $n' \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$ and $n := (\mathsf{nonce}, n')$.

- *Generation:* $n^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Let $n \leftarrow \mathsf{make\_nonce}()$, $n^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (n^{\mathsf{hnd}}, n, \mathsf{nonce}, ())$.

- *Parsing:* $arg \leftarrow \mathsf{parse\_nonce}(n)$.

  If n is of the form (nonce, n') with $n' \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, return $()$, else $\downarrow$.

Signatures

- *Key constructor:* $(sk^*, pk^*) \leftarrow \mathsf{make\_sig\_keypair}()$.

  Let $(sk, pk) \leftarrow \mathsf{gen}_\mathsf{S}(1^k, 1^{\mathsf{max\_skc}(k)})$ and set $sk^* := (\mathsf{sks}, sk, pk)$ and $pk^* := (\mathsf{pks}, pk)$.

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_sig\_keypair}()$.

  Let $(sk^*, pk^*) \leftarrow \mathsf{make\_sig\_keypair}()$, $sk^{\mathsf{hnd}} := curhnd_u{+}{+}$, $pk^{\mathsf{hnd}} := curhnd_u{+}{+}$, $D_u :\Leftarrow (sk^{\mathsf{hnd}}, sk^*, \mathsf{sks}, (0))$, and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, \mathsf{pks}, ())$.[21]

- *Public-key parsing:* $arg \leftarrow \mathsf{parse\_pks}(pk^*)$.

  If pk^* is of the form (pks, pk) with $pk \in \{0,1\}^{\mathsf{pks\_len}'(k)}$, return $()$, else $\downarrow$.

- *Signature constructor:* $s^* \leftarrow \mathsf{make\_sig}(sk^*, l, c)$, with $sk^*, l \in \{0,1\}^+$ and $c \in \mathbb{N}$ with $c \leq \mathsf{max\_skc}(k)$.

  Select $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, set $(sk := sk^*[2]$ and $pk := sk^*[3]$, sign $sig \leftarrow \mathsf{sign}_{sk,c}((r, l))$, and return $s^* := (\mathsf{sig}, pk, r, l, sig)$.

- *Signature generation:* $s^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Parse l^{hnd} if necessary. (Secret keys are not received from the network and thus always parsed.) If $D_u[sk^{\mathsf{hnd}}].type \neq \mathsf{sks}$ or $D_u[sk^{\mathsf{hnd}}].add\_arg[1] \geq \mathsf{max\_skc}(k)$ or $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, then return $\downarrow$. Otherwise set $sk^* := D_u[sk^{\mathsf{hnd}}].word$, $c := D_u[sk^{\mathsf{hnd}}].add\_arg[1]{+}{+}$, $l := D_u[l^{\mathsf{hnd}}].word$, and $s^* \leftarrow \mathsf{make\_sig}(sk^*, l, c)$. If $|s^*| > \mathsf{max\_len}(k)$, decrement $D_u[sk^{\mathsf{hnd}}].add\_arg[1]$ again and return $\downarrow$, else set $s^{\mathsf{hnd}} := curhnd_u{+}{+}$ and $D_u :\Leftarrow (s^{\mathsf{hnd}}, s^*, \mathsf{sig}, ())$.

- *Signature parsing:* $arg \leftarrow \mathsf{parse\_sig}(s^*)$.

  If s^* is not of the form $(\mathsf{sig}, pk, r, l, sig)$ with $pk \in \{0,1\}^{\mathsf{pks\_len}'(k)}$, $r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, $l \in \{0,1\}^+$, and $sig \in \{0,1\}^{\mathsf{sig\_len}'(k,|l|)}$, return $\downarrow$. Also return $\downarrow$ if l is not a tagged list or if $\mathsf{test}_{pk}(sig, (r, l)) = \mathsf{false}$. Otherwise set $pk^* := (\mathsf{pks}, pk)$ and $arg := (pk^*, l)$.

- *Signature verification:* $v \leftarrow \mathsf{verify}(s^{\mathsf{hnd}}, pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Parse s^{hnd} yielding arg. If $D_u[s^{\mathsf{hnd}}].type \neq \mathsf{sig}$, return $\downarrow$. Else let $(pk^*, l) := arg$. Let $v := \mathsf{true}$ if $D_u[pk^{\mathsf{hnd}}].word = pk^*$ and $D_u[l^{\mathsf{hnd}}].word = l$, else $v := \mathsf{false}$.[22]

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_sig}(s^{\mathsf{hnd}})$.

  Parse s^{hnd} yielding arg. If $D_u[s^{\mathsf{hnd}}].type \neq \mathsf{sig}$, return $\downarrow$. Otherwise, let $(pk^{\mathsf{hnd}}, D_u) :\leftarrow (arg[1], \mathsf{pks}, ())$.

[21]The argument 0 of the secret key is the counter and will change.

[22]The actual signature test is already performed when s^{hnd} is parsed. We simply return false if a test is attempted with another public key or message than those contained in the signature.

- *Message retrieval:* $l^{\mathsf{hnd}} \leftarrow \mathsf{msg\_of\_sig}(s^{\mathsf{hnd}})$.

  Parse s^{hnd} yielding *arg*. If $D_u[s^{\mathsf{hnd}}].type \neq \mathsf{sig}$, return $\downarrow$. Otherwise, let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (arg[2], \mathsf{list}, ())$.

Public-key encryption

- *Key constructor:* $(sk^*, pk^*) \leftarrow \mathsf{make\_enc\_keypair}()$.

  Let $(sk, pk) \leftarrow \mathsf{gen_E}(1^k)$ and set $sk^* := (\mathsf{ske}, sk)$ and $pk^* := (\mathsf{pke}, pk)$.

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$.

  Set $(sk^*, pk^*) \leftarrow \mathsf{make\_enc\_keypair}()$, $sk^{\mathsf{hnd}} := curhnd_u\text{++}$, $pk^{\mathsf{hnd}} := curhnd_u\text{++}$, $D_u :\Leftarrow (sk^{\mathsf{hnd}}, sk^*, \mathsf{ske}, ())$, and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, pk^*, \mathsf{pke}, ())$.

- *Public-key parsing:* $arg \leftarrow \mathsf{parse\_pke}(pk^*)$.

  If pk^* is of the form (pke, pk) with $pk \in \{0,1\}^{\mathsf{pke\_len}(k)}$, return $()$, else $\downarrow$.

- *Encryption constructor:* $c^* \leftarrow \mathsf{make\_enc}(pk^*, l)$, for $pk^*, l \in \{0,1\}^+$.

  Let $pk := pk^*[2]$, set $r \stackrel{\mathcal{R}}{\leftarrow} \{0,1\}^{\mathsf{nonce\_len}(k)}$, encrypt $c \leftarrow \mathsf{E}_{pk}((r, l))$, and let $c^* := (\mathsf{enc}, pk, c, r)$.

- *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Parse pk^{hnd} and l^{hnd} if necessary. If $D_u[pk^{\mathsf{hnd}}].type \neq \mathsf{pke}$ or $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, then return $\downarrow$. Otherwise, set $pk^* := D_u[pk^{\mathsf{hnd}}].word$, $l := D_u[l^{\mathsf{hnd}}].word$, and $c^* \leftarrow \mathsf{make\_enc}(pk^*, l)$. If $c^* = \downarrow$ or $|c^*| > \mathsf{max\_len}(k)$ then return $\downarrow$, else set $c^{\mathsf{hnd}} := curhnd_u\text{++}$ and $D_u :\Leftarrow (c^{\mathsf{hnd}}, c^*, \mathsf{enc}, ())$.

- *Ciphertext parsing:* $arg \leftarrow \mathsf{parse\_enc}(c^*)$.

  If c^* is not of the form (enc, pk, c, r) with $pk \in \{0,1\}^{\mathsf{pke\_len}(k)}$, $c \in \{0,1\}^+$, and $r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, return $\downarrow$, else $arg := (pk^*) := ((\mathsf{pke}, pk))$.

- *Functional decryption:* $l \leftarrow \mathsf{parse\_decrypt}(sk^*, c^*)$.

  Let $sk := sk^*[2]$, $c := c^*[3]$, and $r := c^*[4]$. Decrypt $l^* := \mathsf{D}_{sk}(c)$. If $l^* = \downarrow$ or $l^*[1] \neq r$ or $l^*[2]$ is not a tagged list, return $l := \downarrow$, else $l := l^*[2]$.

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  Parse c^{hnd} yielding $arg =: (pk^*)$. Return $\downarrow$ if $D_u[sk^{\mathsf{hnd}}].type \neq \mathsf{ske}$ or $D_u[c^{\mathsf{hnd}}].type \neq \mathsf{enc}$ or $D_u[sk^{\mathsf{hnd}} + 1].word \neq pk^*$.[23] Otherwise, set $sk^* := D_u[sk^{\mathsf{hnd}}].word$, $c^* := D_u[c^{\mathsf{hnd}}].word$, and $l \leftarrow \mathsf{parse\_decrypt}(sk^*, c^*)$. If $l = \downarrow$, return $\downarrow$. Otherwise let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_enc}(c^{\mathsf{hnd}})$.

  Parse c^{hnd} yielding *arg*. If $D_u[c^{\mathsf{hnd}}].type \neq \mathsf{enc}$, return $\downarrow$. Otherwise let $(pk^{\mathsf{hnd}}, D_u) :\leftarrow (arg[1], \mathsf{pke}, ())$.

5.5.4 Send Commands

- $\mathsf{send\_}x(v, l^{\mathsf{hnd}})$, for $x \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}\}$ and $v \in \{1, \ldots, n\}$.

  Parse l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$, output $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,x}!$.

[23] This means that one never tries to decrypt with respect to another public key than the one contained in the ciphertext.

5.5.5 Network Inputs

- *Network input:* On input l at a port $\mathsf{net}_{w,u,x}?$, for $l \in \{0,1\}^+$ and $|l| \leq \mathsf{max\_len}(k)$.

  Test if $l = (\mathsf{list}, x_1, \ldots, x_j)$ for some $j \in \mathbb{N}_0$ and values $x_i \in \{0,1\}^+$. If yes, let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$ and output (w, x, l^{hnd}) at $\mathsf{out}_u!$.

5.6 Properties of the Real System

The following properties of the real system are useful:

Lemma 5.2 The real systems $Sys^{\mathsf{cry},\mathsf{real},x}_{n,\mathcal{S},\mathcal{E},L'}$ for $x \in \{\mathsf{stan}, \mathsf{loc}\}$ have the following properties:

1. The argument hnd_u of D_u is a key attribute.
2. The only modifications to existing entries x in D_u are that $x.type$ changes from null to something else and that $x.add\_arg$ changes from $()$ to something else at the same time, and counter updates in entries of type sks.
3. At all times we have $|x.word| \leq \mathsf{max\_len}(k)$ for all $x \in D_u$, except possibly if $x.type \in \{\mathsf{sks}, \mathsf{ske}\}$.
4. The systems are polynomial-time.

□

Proof.

1. This was already shown in Section 5.4.2.
2. This can easily be seen by inspection of the commands.
3. By Part 2, we only need to consider commands that produce new entries. In most cases, the subroutine "$(x^{\mathsf{hnd}}, D_u) :\leftarrow \ldots$" is used, which tests for this bound. For signatures and encryptions, the outer command contains the test. For nonces and public keys, their lengths are $\mathsf{nonce\_len}^*(k)$, $\mathsf{pks\_len}^*(k)$, and $\mathsf{pke\_len}^*(k)$, respectively, which are bounded by $\mathsf{max\_len}(k)$ by Section 5.3.
4. We have to show that each M_u only accepts a polynomial number of polynomial-length inputs; the action in each command is then clearly polynomial-time because all basic algorithms used are this. For the numbers, this holds because there is a counter $steps_{\mathsf{p}?}$ for inputs at each port $\mathsf{p}?$, each with the polynomial bound $\mathsf{max\_in}(k)$. As each input adds at most 2 entries to D_u, the dynamic bound $curhnd_u$ for handle parameters is also polynomially bounded. All other input parameters have static polynomial domain restrictions. Hence the length bounds on inputs (recall Section 4.5.1) are indeed polynomial in k.

■

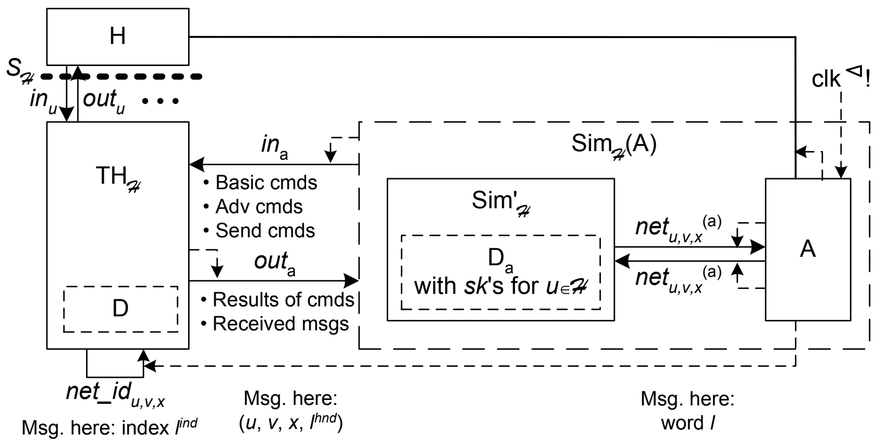

Figure 4: Set-up of the simulator. User in- and outputs are scheduled as in the ideal system.

6 Simulator

We claim that every real system $Sys^{\mathsf{cry,real},x}_{n,\mathcal{S},\mathcal{E},L'}$ for $x \in \{\mathsf{stan}, \mathsf{loc}\}$ in Section 5 is as secure as the ideal system $Sys^{\mathsf{cry,id},x}_{n,L}$ in Section 4 for the parameters $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$, even with a blackbox simulation. The main step in this proof is to construct a simulator $\mathsf{Sim}_{\mathcal{H}}$ for each set $\mathcal{H}$ such that for every real adversary A, the combination $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ of $\mathsf{Sim}_{\mathcal{H}}$ and A achieves the same effects in the ideal system as the adversary A in the real system, see Figure 4.

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 $\mathsf{in_a}?$ and vice versa, see Figure 4. In both directions, $\mathsf{Sim}_{\mathcal{H}}$ has to parse messages completely because it can only construct the other version (abstract or real) bottom-up. This is done by recursive algorithms. In some cases, the simulator cannot produce any corresponding message. We collect these cases in so-called *error sets* and show later that they cannot occur at all or only with negligible probability.

6.1 Ports and Scheduling

The ports of $\mathsf{Sim}_{\mathcal{H}}$ are shown in Figure 4. In particular, the clock ports, i.e., the scheduling, are as follows: A remains the master scheduler. It also still schedules the connections between itself and H, the network connections $net_{u,v,x}$ with $(u, v, x) \notin ch\_honest$ (including the doubled connections $net^{\mathsf{a}}_{u,v,x}$), and in the stand-alone version the user in- and outputs, i.e., these clock ports of A are unchanged. Each clock port $\mathsf{net}_{u,v,x}{}^{\lhd}!$ with $(u, v, x) \in ch\_honest$ is renamed into $\mathsf{net\_id}_{u,v,x}{}^{\lhd}!$ so that A also schedules the ideal secure channels. (Port renaming is allowed in blackbox reductions, cf. [34].) Recall that $\mathsf{TH}_{\mathcal{H}}$ schedules its outputs at $\mathsf{out_a}!$.

Thus $\mathsf{Sim}_{\mathcal{H}}$ mainly schedules the connection in_{a}. It immediately does this (by outputting 1 at $\mathsf{in_a}{}^{\lhd}!$), whenever it makes an output at $\mathsf{in_a}!$; we do not repeat this below. For local commands, we abbreviate subroutine behavior as follows:

"Call $y \leftarrow x$ at $\mathsf{in_a}!$, expecting ...",

means that $\mathsf{Sim}_{\mathcal{H}}$ outputs x at port $\mathsf{in_a}!$ and schedules it, waits for an input at $\mathsf{out_a}?$ by setting all other length functions to 0, and assigns the input to a variable y. If y is not in the domain given after "expecting", $\mathsf{Sim}_{\mathcal{H}}$ aborts all current recursive algorithms and its outermost transition. As $\mathsf{TH}_{\mathcal{H}}$ reacts to local commands with an output at $\mathsf{out_a}!$ and schedules it immediately, we really get subroutine behavior.

6.2 Long-term 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 $\mathsf{p}?$.

6.2.1 Database D_{a}

To ensure that the same abstract message is always mapped to the same real message and vice versa, $\mathsf{Sim}_{\mathcal{H}}$ maintains a database D_{a} of already mapped adversary handles. Each entry in D_{a} has attributes

$$(hnd_{\mathsf{a}}, word, add\_arg).$$

For each tuple $x \in D_{\mathsf{a}}$:

- $x.hnd_{\mathsf{a}} \in \mathcal{HNDS}$ is used as the primary key attribute. However, its use is not as straightforward as in the ideal and real system, compare Lemma 6.2.
- $x.word \in \{0,1\}^*$ is the real representation of x, or ϵ for unknown secret keys (of the adversary).
- $x.add\_arg$ is a list of additional arguments. Typically it is $()$. However, for public keys received from the adversary it is (adv), and for keys from honest users, where the simulator generated a key pair, it is of the form (honest, sk^*).

6.2.2 Conventions about Handles

Initially, D_{a} is empty. $\mathsf{Sim}_{\mathcal{H}}$ keeps a counter $curhnd_{\mathsf{a}}$ of type $\mathcal{HNDS}$, which denotes the current size of D_{a}, except temporarily within the algorithm $\mathsf{id2real}$ defined below.

6.2.3 Input Counters

For each of its input ports $\mathsf{p}?$, $\mathsf{Sim}_{\mathcal{H}}$ maintains a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ initialized with 0 for the number of inputs at that port. The corresponding bounds $bound_{\mathsf{p}?}$ are $\mathsf{max\_in}(k)$ for the network ports and $\mathsf{max\_in}_{\mathsf{a}}(k) = 10n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ for $\mathsf{out}_{\mathsf{a}}?$.

6.3 Input Evaluation

6.3.1 General Conventions

Upon each input at a port $\mathsf{p}?$, the main simulator $\mathsf{Sim}_{\mathcal{H}}$ first increments the counter $steps_{\mathsf{p}?}$. This is the only case where such a counter is increased. Once a counter $steps_{\mathsf{p}?}$ has reached $bound_{\mathsf{p}?}$, the length function for this port is always zero.

Otherwise, the length functions are determined by the domain specified for each input below.

6.3.2 Send Commands

In this section, we define the behavior of $\mathsf{Sim}_{\mathcal{H}}$ when it receives an input from $\mathsf{TH}_{\mathcal{H}}$. We only consider "unsolicited" inputs here, not the immediate results of local commands. These inputs are the results of send commands by honest users.

- *Input from* $\mathsf{TH}_{\mathcal{H}}$*:* On input m at port $\mathsf{out}_{\mathsf{a}}?$, for $m = (u, v, x, l^{\mathsf{hnd}})$ with $(u, v, x) \in ch\_to\_adv$ and $l^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k) := 6n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$.

  If $D_{\mathsf{a}}[l^{\mathsf{hnd}}] \neq \downarrow$, let $l := D_{\mathsf{a}}[l^{\mathsf{hnd}}].word$. Else set $curhnd_{\mathsf{a}}$++ and produce such a word with an algorithm $l \leftarrow \mathsf{id2real}(l^{\mathsf{hnd}})$ (with side-effects). Then output l at port $\mathsf{net}_{u,v,x}!$.

We will show that $\mathsf{max\_hnd}(k)$ is an upper bound on the database size in $\mathsf{TH}_\mathcal{H}$ if $\mathsf{TH}_\mathcal{H}$ interacts with $\mathsf{Sim}_\mathcal{H}$. Actually $\mathsf{TH}_\mathcal{H}$ always uses $l^\mathsf{hnd} = \mathit{curhnd}_\mathsf{a}$++ for new adversary handles. Hence $\mathsf{Sim}_\mathcal{H}$'s update of $\mathit{curhnd}_\mathsf{a}$ restores "correct derivation".

Overall structure of id2real The algorithm id2real recursively parses an abstract message, builds up a corresponding real message, and enters all new message parts into D_a. It only interacts with $\mathsf{TH}_\mathcal{H}$ by local commands, so that we get an uninterrupted dialogue between $\mathsf{Sim}_\mathcal{H}$ and $\mathsf{TH}_\mathcal{H}$.

The domain expectations of id2real and certain additional format claims made will be proven in Section 7 for interaction with $\mathsf{TH}_\mathcal{H}$. The main explanation is that id2real is only called for new handles. They only occur if the corresponding entry in $\mathsf{TH}_\mathcal{H}$ was constructed at the request of an honest user, and thus by a basic command. Hence none of the exceptional data formats occurs.

Now $\mathsf{id2real}(m^\mathsf{hnd})$ is defined as follows.

1. Call $(\mathit{type}, (m_1^\mathsf{hnd}, \ldots, m_j^\mathsf{hnd})) \leftarrow \mathsf{adv\_parse}(m^\mathsf{hnd})$ at $\mathsf{in_a}!$ (we use the superscript $^\mathsf{hnd}$ although not all ideal attributes are handles), expecting $\mathit{type} \in \mathit{typeset} \setminus \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ 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 $|m_i^\mathsf{hnd}| \leq \mathsf{max\_len}(k)$.

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

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 $m_i := D_\mathsf{a}[m_i^\mathsf{hnd}].\mathit{word}$. Else make a recursive call $m_i \leftarrow \mathsf{id2real}(m_i^\mathsf{hnd})$. Let $\mathit{arg}^\mathsf{real} := (m_1, \ldots, m_j)$.

4. Construct and enter the real message m depending on type:

   - If $\mathit{type} \in \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}\}$, set $m \leftarrow \mathsf{make\_}\mathit{type}(\mathit{arg}^\mathsf{real})$ and $D_\mathsf{a} :\Leftarrow (m^\mathsf{hnd}, m, ())$.
   - If $\mathit{type} \in \{\mathsf{pks}, \mathsf{pke}\}$, call $(\mathit{sk}^*, \mathit{pk}^*) \leftarrow \mathsf{make\_sig\_keypair}()$ or $(\mathit{sk}^*, \mathit{pk}^*) \leftarrow \mathsf{make\_enc\_keypair}()$, respectively, and set $m := \mathit{pk}^*$ and $D_\mathsf{a} :\Leftarrow (m^\mathsf{hnd}, m, (\mathsf{honest}, \mathit{sk}^*))$.
   - If $\mathit{type} = \mathsf{sig}$, we claim that $\mathit{arg}^\mathsf{real}$ is of the form (pk^*, l, c) with $c \in \mathbb{N}$. Let $\mathit{pk}^\mathsf{hnd} := m_1^\mathsf{hnd}$. We claim that $D_\mathsf{a}[\mathit{pk}^\mathsf{hnd}].\mathit{add\_arg}$ is of the form $(\mathsf{honest}, \mathit{sk}^*)$. Then set $m \leftarrow \mathsf{make\_sig}(\mathit{sk}^*, l, c)$ and $D_\mathsf{a} :\Leftarrow (m^\mathsf{hnd}, m, ())$.
   - If $\mathit{type} = \mathsf{enc}$, we claim that $l^\mathsf{hnd} := m_2^\mathsf{hnd} \neq \downarrow$, and distinguish two cases: If $l^\mathsf{hnd} \in \mathcal{HNDS}$ (i.e., a cleartext handle, not only a length was output), let $m \leftarrow \mathsf{make\_enc}(\mathit{arg}^\mathsf{real})$ and $D_\mathsf{a} :\Leftarrow (m^\mathsf{hnd}, m, ())$.

     Otherwise $\mathit{arg}^\mathsf{real}$ is of the form $(\mathit{pk}^*, \mathit{len})$ with $\mathit{len} \in \mathbb{N}$. Then $\mathsf{Sim}_\mathcal{H}$ encrypts a fixed message of the correct length; it must not be a list. Let $\mathit{pk} := \mathit{pk}^*[2]$ and $\mathit{len}^* := \mathsf{list\_len}(\mathsf{nonce\_len}(k), \mathit{len})$. Encrypt $c \leftarrow \mathsf{E}_{\mathit{pk}}(1^{\mathit{len}^*})$ and set $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $m := (\mathsf{enc}, \mathit{pk}, c, r)$, and $D_\mathsf{a} :\Leftarrow (m^\mathsf{hnd}, m, ())$.

6.3.3 Network Inputs

Now we define the behavior of $\mathsf{Sim}_\mathcal{H}$ when it receives an input from A.

- *Network input:* On input l at a port $\mathsf{net}_{w,u,x}?$ with $(w, u, x) \in \mathit{ch\_from\_adv}$, for $l \in \{0,1\}^+$ and $|l| \leq \mathsf{max\_len}(k)$.

  If l is not a tagged list, abort the transition. Otherwise translate l into a corresponding handle l^hnd by an algorithm $l^\mathsf{hnd} \leftarrow \mathsf{real2id}(l)$ (with side-effects). Then output the command $\mathsf{adv\_send\_}x(w, u, l^\mathsf{hnd})$ at port $\mathsf{in_a}!$.

Overall structure of $\mathsf{real2id}$ The algorithm $\mathsf{real2id}$ recursively parses a real message, builds up a corresponding term in $\mathsf{TH}_{\mathcal{H}}$, and enters all subterms into D_{a}. It only interacts with $\mathsf{TH}_{\mathcal{H}}$ by local commands. As these have a subroutine behavior, we get an uninterrupted dialogue between $\mathsf{Sim}_{\mathcal{H}}$ and $\mathsf{TH}_{\mathcal{H}}$.

- $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$, for $m \in \{0,1\}^+$.

  If there is already a handle m^{hnd} with $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, return that. Otherwise set $(type, arg) := \mathsf{parse}(m)$ and call a type-specific algorithm $add\_arg \leftarrow \mathsf{real2id\_}type(m, arg)$ (with side-effects). Set $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$.

In the definitions of $\mathsf{real2id\_}type$, we write the parameter arg in the form that it must have when parse returned $(type, arg)$. Each $\mathsf{real2id\_}type$ ignores the results of its calls to $\mathsf{TH}_{\mathcal{H}}$, which are predictable handles. Thus the length function for $\mathsf{out_a}$? can be 1 in those states.

Garbage $add\_arg \leftarrow \mathsf{real2id\_garbage}(m, ())$. Call $m^{\mathsf{hnd}} \leftarrow \mathsf{adv\_garbage}(|m|)$ at $\mathsf{in_a}$!. Then return $()$.

Data $add\_arg \leftarrow \mathsf{real2id\_data}(m, (m'))$. Call $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m')$ at $\mathsf{in_a}$!. Return $()$.

Lists $add\_arg \leftarrow \mathsf{real2id\_list}(m, (m_1, \ldots, m_j))$. Make recursive calls $m_i^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m_i)$ for $i := 1, \ldots, j$ (in this order). Then call $m^{\mathsf{hnd}} \leftarrow \mathsf{list}(m_1^{\mathsf{hnd}}, \ldots, m_j^{\mathsf{hnd}})$ at $\mathsf{in_a}$!. Return $()$.

Nonces $add\_arg \leftarrow \mathsf{real2id\_nonce}(m, ())$. Call $m^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$ at $\mathsf{in_a}$!. Return $()$.

Signatures

- $add\_arg \leftarrow \mathsf{real2id\_pks}(m, ())$.

  Call $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_sig\_keypair}()$ at $\mathsf{in_a}$!. Set $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}{++}, \epsilon, ())$ for the secret key and return $add\_arg := (\mathsf{adv})$ for the public key.[24]

- $add\_arg \leftarrow \mathsf{real2id\_sig}(s^*, (pk^*, l))$.

  Make recursive calls $pk^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(pk^*)$ and $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$.

  If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{adv})$, set $sk^{\mathsf{hnd}} := pk^{\mathsf{hnd}} - 1$. Then call $s^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}$! and return $()$.[25]

  Otherwise, let $(\mathsf{sig}, pk, r, l, sig) := s^*$. Verify whether the adversary has seen another signature on the same message, i.e., if there exist s^{hnd}, r', and sig' with $D_{\mathsf{a}}[s^{\mathsf{hnd}}].word = (\mathsf{sig}, pk, r', l, sig')$ and $D_{\mathsf{a}}[s^{\mathsf{hnd}}].type = \mathsf{sig}$. If yes, call $t^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_sig}(s^{\mathsf{hnd}})$ at $\mathsf{in_a}$! and return $()$.

  Otherwise, $\mathsf{Sim}_{\mathcal{H}}$ gives up the simulation.[26]

[24]Here m is a new public key from the adversary. $\mathsf{TH}_{\mathcal{H}}$ sets $sk^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++; $pk^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++, while $\mathsf{Sim}_{\mathcal{H}}$ does not know a real secret key. $\mathsf{Sim}_{\mathcal{H}}$ therefore makes an "empty" entry in D_{a} for the secret key, while the public key is entered in D_{a} by the last step of $\mathsf{real2id}$ as for all other data types.

[25]$\mathsf{TH}_{\mathcal{H}}$ typically uses another counter value c for this signature than that used in the real adversary signature. Honest users cannot notice this because there is no basic command to retrieve c. Neither can A retrieve c with $\mathsf{adv\_parse}$, because $\mathsf{Sim}_{\mathcal{H}}$ only uses that command when constructing signatures it has not seen before.

[26]This is the case of a forged signature on a new message, and no command of $\mathsf{TH}_{\mathcal{H}}$ allows that. It is easy to see that an adversary against $\mathsf{Sim}_{\mathcal{H}}$ indeed forged a signature here. However, in the real system such a signature may already exist, but not be available to the adversary. Thus the proof that this case is cryptographically impossible is more complicated; see Section 7.7.6.

Public-key encryption

- $add\_arg \leftarrow \mathsf{real2id\_pke}(m, ())$.

  Call $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$ at $\mathsf{in_a}!$. Set $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}\text{++}, \epsilon, ())$ for the secret key and return $add\_arg := (\mathsf{adv})$ for the public key.

- $add\_arg \leftarrow \mathsf{real2id\_enc}(c^*, (pk^*))$.

  Make a recursive call $pk^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(pk^*)$. Now, to produce a corresponding abstract ciphertext in $\mathsf{TH}_{\mathcal{H}}$, we try to find out the message.

  If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{adv})$, then $\mathsf{Sim}_{\mathcal{H}}$ does not know the corresponding secret key. Call $c^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_ciph}(pk^{\mathsf{hnd}}, |c^*|)$ at $\mathsf{in_a}!$ and return $()$.[27]

  If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk^*)$, let $l \leftarrow \mathsf{parse\_decrypt}(sk^*, c^*)$. If $l = \downarrow$, call $c^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_ciph}(pk^{\mathsf{hnd}}, |c^*|)$ at $\mathsf{in_a}!$ and return $()$. Else make a recursive call $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$, call $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$, and return $()$.

6.4 Properties of the Simulator

We first show that the simulator is valid.

Lemma 6.1 Each machine $\mathsf{Sim}_{\mathcal{H}}$ is polynomial-time. Hence for every polynomial-time adversary A on the real system, the joint machine $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ is polynomial-time and thus a valid adversary on the ideal system. □

Proof. For each of its input ports, $\mathsf{Sim}_{\mathcal{H}}$ has a polynomially bounded input counter and polynomially bounded length functions. Further, every transition is polynomial-time because an execution of $\mathsf{id2real}$ without inner recursions is clearly polynomial-time, and every inner recursion starts with an output at $\mathsf{in_a}!$ and thus ends a transition. The combination $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ of two polynomial-time machines $\mathsf{Sim}_{\mathcal{H}}$ and A is clearly polynomial-time again. ■

The following properties of the simulator are useful for the correctness proof.

Lemma 6.2 Each machine $\mathsf{Sim}_{\mathcal{H}}$ has the following properties:

1. Each call $\mathsf{id2real}(m^{\mathsf{hnd}})$ leads to $D_{\mathsf{a}}[m^{\mathsf{hnd}}] \neq \downarrow$ unless the execution aborts.
2. If $\mathsf{Sim}_{\mathcal{H}}$ interacts with $\mathsf{TH}_{\mathcal{H}}$, the following holds for the algorithm $\mathsf{id2real}$:
   - Calls $\mathsf{id2real}(m^{\mathsf{hnd}})$ are only made for $m^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$, and at most one such call for each m^{hnd}.
   - At most $\mathsf{max\_hnd}(k)$ outputs at $\mathsf{in_a}!$ are made.
   - No new entries in D (of $\mathsf{TH}_{\mathcal{H}}$) are made.
3. Each call $\mathsf{real2id}(m)$ leads to a recursion tree with the following properties:
   - At most $|m|-1$ additional calls $\mathsf{real2id}(m_i)$, each with $|m_i| > 0$ are made.
   - At most $|m|$ outputs at are made at $\mathsf{in_a}!$.
   - $curhnd_{\mathsf{a}}$ is increased by at most $2|m|$, and thus at most $2|m|$ new entries are made in D_{a}.

[27] In the real system, this ciphertext is not necessarily invalid. However, it was made by A with its own public key. (If it had been produced by an honest participant, $\mathsf{Sim}_{\mathcal{H}}$ would have first entered it into D_{a} upon a send command.) Thus no real machine M_v will try to decrypt it, so that the honest users do not notice the difference. Further, if they send it back to A, then $\mathsf{Sim}_{\mathcal{H}}$ sends the original.

4. If $\mathsf{Sim}_\mathcal{H}$ interacts with $\mathsf{TH}_\mathcal{H}$, the following holds:
    - The argument hnd_a of D_a is a key attribute.
    - Outside any execution of $\mathsf{id2real}$, the entries in D_a are consecutively numbered, i.e., $\{x.hnd_\mathsf{a} \mid x \in D_\mathsf{a}\} = \{1, \ldots, curhnd_\mathsf{a}\}$.
5. In interaction of $\mathsf{TH}_\mathcal{H}$ and $\mathsf{Sim}_\mathcal{H}$, the following holds:
    - No handle output by $\mathsf{TH}_\mathcal{H}$ is rejected by $\mathsf{Sim}_\mathcal{H}$.
    - 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.

□

Proof.

1. Unless an execution of $\mathsf{id2real}(m^{\mathsf{hnd}})$ aborts, it ends in Step 4 with an assignment $D_\mathsf{a} :\Leftarrow (m^{\mathsf{hnd}}, \ldots)$.
2. Calls $\mathsf{id2real}(m^{\mathsf{hnd}})$ are only made by the outer transition for send commands and in Step 3 of $\mathsf{id2real}$. Both times it was verified that $m^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ and $D_\mathsf{a}[m_i^{\mathsf{hnd}}] \neq \downarrow$. We want to show that every m^{hnd} is used in at most one call.

    By Part 1, this can only be violated if a first call $\mathsf{id2real}(m^{\mathsf{hnd}})$ has not finished when a second one is made, i.e., the second call is a node in the recursion tree of the first one. In a recursion tree, $\mathsf{Sim}_\mathcal{H}$ only obtains new handles with the command $\mathsf{adv\_parse}$. Hence $\mathsf{TH}_\mathcal{H}$ only follows the components of a term. By Lemma 4.1.2, all components of a term in $\mathsf{TH}_\mathcal{H}$ have smaller indices. In particular, no index and thus no adversary handle occurs twice, which proves our claim.

    The other bound follows immediately because each subcall (without inner recursions) makes one output at $\mathsf{in_a}!$ (in Step 1).

    As the only command to $\mathsf{TH}_\mathcal{H}$ is $\mathsf{adv\_parse}$, no new entries in D are made.
3. We consider a call $\mathsf{real2id}(m)$ with $L := |m|$. Once we have shown the bound on the number of recursive calls, the other bounds follow immediately because each subcall (without inner recursions) makes at most one output at $\mathsf{in_a}!$ and increases $curhnd_\mathsf{a}$ by at most 2.

    We show the first claim inductively over L. Subcalls only happen when parsing results in $type \in \{\mathsf{list}, \mathsf{sig}, \mathsf{enc}\}$. The basis $L = 1$ is clear because no subcall happens for a message of length 1. Now let $L > 1$.

    For lists, we have $m = (\mathsf{list}, m_1, \ldots, m_j)$ and all m_i are non-empty. By the preconditions on $\mathsf{list\_len}$ we have $|m_i| < L$ for all i. Hence we can use the induction hypothesis for each m_i. This immediately gives that all subcalls are made for non-empty messages, and the number of subcalls is bounded by $j + (|m_1| - 1) + \ldots + (|m_j| - 1) = |m_1| + \ldots + |m_j| \leq L - 1$.

    For signatures, we have $m = s^* = (\mathsf{sig}, pk, r, l, sig)$ and subcalls are made for $pk^* = (\mathsf{pks}, pk)$ and l. Clearly $|pk^*| > 0$, and $|l| > 0$ was verified in parsing. Parsing pk^* yields $type = \mathsf{pks}$; thus no further subcalls are made for it. Further, $|l| \leq L - 4$ because all components are non-empty. By the induction hypothesis, at most $L - 5$ subcalls are made for it, and thus overall at most $L - 3$.

    For ciphertexts, we have $m = c^* = (\mathsf{enc}, pk, c, r)$, and subcalls are made for $pk^* = (\mathsf{pks}, pk)$ and possibly $l := l^*[2]$, where l^* is the result of decrypting with the encryption machine and $|l^*| \leq |c|$ was verified. Clearly $|pk^*| > 0$, and $|l| > 0$ by a test that l is a tagged list before the recursive call. Parsing pk^* yields $type = \mathsf{pke}$; thus no further subcalls are made for it. Further, $|c| \leq L - 3$ because all components are non-empty, and thus also $|l| \leq L - 3$. By the induction hypothesis, at most $L - 3$ subcalls are made for it, and thus overall at most $L - 2$.

4. We show these conditions on handles inductively over the inputs.

   Upon network inputs, new entries in D_a are always made with the handle $curhnd_\mathsf{a}$++. The first two claims follow immediately.

   Upon inputs from $\mathsf{TH}_\mathcal{H}$, $curhnd_\mathsf{a}$ is always immediately updated when $\mathsf{TH}_\mathcal{H}$ sends a new handle; thus $curhnd_\mathsf{a}$ equals $\mathsf{TH}_\mathcal{H}$'s variable $curhnd_\mathsf{a}$, for which the claim holds with $\mathsf{TH}_\mathcal{H}$'s handles. Further, $\mathsf{TH}_\mathcal{H}$ immediately outputs each handle (Lemma 4.2.4), and by Part 1 each call $\mathsf{id2real}(m^\mathsf{hnd})$ ends with a new entry x with $x.hnd_\mathsf{a} = m^\mathsf{hnd}$. Hence at the end of an outermost execution of $\mathsf{id2real}$ the claim also holds for the handles in D_a.

5. In the given combination, the free ports are $\mathsf{in}_u?$ for $u \in \mathcal{H}$ and $\mathsf{net}_{w,u,x}?$ with $(w, u, x) \in ch\_from\_adv$ and the clock ports for the secure channels (recall Figure 4). We first consider the effects of inputs there separately.

   - User inputs. There are at most n user ports, each accepting at most $\mathsf{max\_in}(k)$ inputs.

     These inputs directly lead to at most $2n\mathsf{max\_in}(k)$ new entries in D.

     Further, each such input leads to at most one direct output at $\mathsf{out}_\mathsf{a}!$ (if it is a send command), which leads to one call $\mathsf{id2real}(m^\mathsf{hnd})$. These calls lead to at most $\mathsf{max\_hnd}(k)$ outputs at $\mathsf{in}_\mathsf{a}!$ and thus at most $\mathsf{max\_hnd}(k)$ responses at $\mathsf{out}_\mathsf{a}!$, whose consequences we have already covered by considering the entire executions of $\mathsf{id2real}$, and to no new entries in D.
   - Network inputs. There are at most $2n^2$ network ports, each accepting at most $\mathsf{max\_in}(k)$ inputs of length at most $\mathsf{max\_len}(k)$. Each input m leads to at most one direct output at $\mathsf{in}_\mathsf{a}!$ (the final ideal message) and a call $\mathsf{real2id}(m)$. This call leads to at most $\mathsf{max\_len}(k) - 1$ further outputs at $\mathsf{in}_\mathsf{a}!$. Overall this gives at most

     $$\mathsf{max\_in_a\_net}(k) := 2n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$$

     outputs at $\mathsf{in}_\mathsf{a}!$. This also gives $\mathsf{max\_in_a\_net}(k)$ responses at $\mathsf{out}_\mathsf{a}!$, whose consequences we have already covered by considering the entire execution of $\mathsf{real2id}(m)$.

     Hence in $\mathsf{TH}_\mathcal{H}$, these inputs lead to at most $2\mathsf{max\_in_a\_net}(k)$ new entries in D.
   - Ideal secure channels. Inputs at a port $\mathsf{net\_id_{u,v,x}}?$ in $\mathsf{TH}_\mathcal{H}$ neither lead to new entries in D nor to outputs at $\mathsf{out}_\mathsf{a}!$.

   Altogether, we get at most

   $$2n\mathsf{max\_in}(k) + 2\mathsf{max\_in_a\_net}(k)$$

   entries in D (independent of how we chose the bounds for the inner ports), and this bounds $curhnd_\mathsf{a}$ in $\mathsf{TH}_\mathcal{H}$. Thus $\mathsf{Sim}_\mathcal{H}$'s handle bound $\mathsf{max\_hnd}(k) = 6n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$ is safe.

   The outputs at $\mathsf{out}_\mathsf{a}!$ and $\mathsf{in}_\mathsf{a}!$ are bounded by, respectively,

   $$n\mathsf{max\_in}(k) + \mathsf{max\_hnd}(k) + \mathsf{max\_in_a\_net}(k);$$

$$\mathsf{max\_hnd}(k) + \mathsf{max\_in_a\_net}(k).$$

Both are smaller than the value $\mathsf{max\_in_a}(k) = 10n^2\mathsf{max\_in}(k)\mathsf{max\_len}(k)$, which is used as $\mathit{bound}_{\mathsf{in_a}}?$ by $\mathsf{TH}_{\mathcal{H}}$ and as $\mathit{bound}_{\mathsf{out_a}}?$ and $\mathit{bound}_{\mathsf{out}_{\mathsf{enc},u}}?$ by $\mathsf{Sim}_{\mathcal{H}}$. This proves the remaining claims.

■

7 Security Proof

In the following, we omit the index $x \in \{\mathsf{stan}, \mathsf{loc}\}$ (stand-alone or localized). The theorem and proof hold for both versions.

Theorem 7.1 *(Security of Cryptographic Library)* Let a secure signature scheme $\mathcal{S}$ and a secure encryption scheme $\mathcal{E}$ be given. Then for all $n \in \mathbb{N}$, all correct parameters L', and all $\mathcal{H} \subseteq \{1, \ldots, n\}$, there is a simulator $\mathsf{Sim}_{\mathcal{H}}$ such that for all polynomial-time honest users H and adversaries A, the view of H while interacting with the correct machines $\mathsf{M}_{u,\mathcal{H}}$ (i.e., the derivations according to the channel model) for $u \in \mathcal{H}$ and A is polynomially indistinguishable from the view of H while interacting with $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ for the parameters $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$.

Using the notation of [34]: For all n and L' and for $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, L')$, we have

$$Sys^{\mathsf{cry,real}}_{n,\mathcal{S},\mathcal{E},L'} \geq^{f,\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{cry,id}}_{n,L}$$

for the canonical mapping f with blackbox simulatability. □

We first give an outline of the proof, see Figure 5.

- *Introducing encryption machines:* We present the two encryption machines $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ of [34], which will later be used to answer encryption and decryption request of the system. Roughly, the first system calculates the correct encryption of a message m, whereas the second one always encrypts the fixed message $1^{|m|}$ and answers decryption requests for ciphertexts generated that way by table look-up. It was shown in [34] that the system $\mathsf{Enc}_{\mathcal{H}}$ is at least as secure as the system $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.

- *Rewriting the real system:* We rewrite the machines M_u such that they use the encryption system $\mathsf{Enc}_{\mathcal{H}}$ (Step 1 in Figure 5). This yields modified machines M'_u. We then replace $\mathsf{Enc}_{\mathcal{H}}$ by its idealized counterpart $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ (Step 2 in Figure 5) and use a composition theorem to show that the derived system is as secure as the original system.

- *Combined system:* We want to prove that the modified real system is at least as secure as the ideal system consisting of the machine $\mathsf{TH}_{\mathcal{H}}$. We use the simulator from Section 6. More precisely, we compare the combination $\mathsf{M}_{\mathcal{H}}$ of the machines M'_u and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ with the combination $\mathsf{THSim}_{\mathcal{H}}$ of the machines $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$. However, there is no direct invariant mapping between the states of these two joint machines. We solve this problem by introducing an intermediate system $\mathsf{C}_{\mathcal{H}}$ with a combined state space (Step 3 in Figure 5).

- *Bisimulations with error sets:* We show that the joint view of H and A, together with the buffers leading to and from them, is equal in interaction with the combined machine $\mathsf{C}_{\mathcal{H}}$ and the two machines $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$, except for certain runs, which we collect in so-called *error sets*. We make this comparison by performing two bisimulation simultaneously (Step 4 in Figure 5). Transitivity and symmetry of indistinguishability then yield the desired result for $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$.

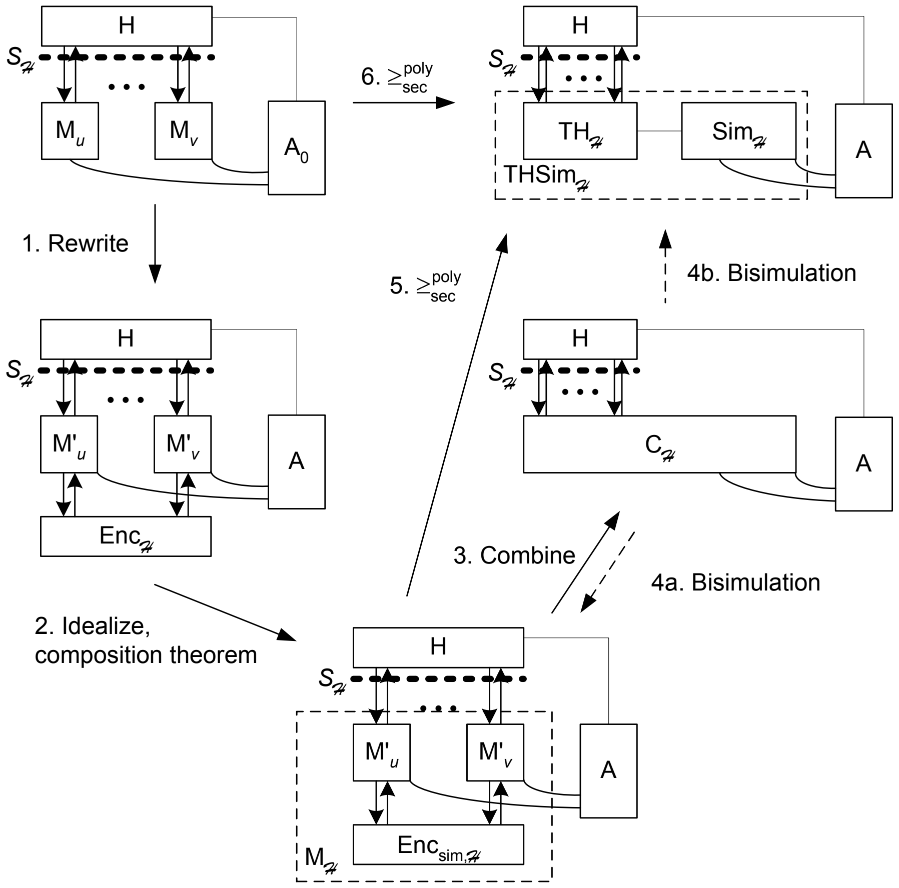

Figure 5: Overview of the proof of Theorem 7.1.

- *Reduction proofs against the underlying cryptography:* We finally show that the aggregated probability of the runs in error sets is negligible as we could otherwise break the underlying cryptography. I.e., we perform reduction proofs against the security definitions of the primitives. This proves that $\mathsf{M}_{\mathcal{H}}$ is (computationally) as secure as the ideal system (Step 5 in Figure 5).

Finally, simulatability is transitive. Hence also the original real system is as secure as the ideal system. The reason for treating encryption separately at the beginning (and not in the final reduction proofs like the other primitives), although every additional machine makes a bisimulation more tedious, is that computational indistinguishability of real and simulated ciphertexts does not give us a mapping that is almost everywhere correct. In the following, we present the proof in detail.

7.1 Introducing Encryption Machines

First we want to replace the encryption of each message m sent among honest users by an encryption of the fixed message $1^{|m|}$, and the decryption of each ciphertext generated that way by table look-up.

7.1.1 Encryption Machines

We use the two encryption systems $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ from [34]; we only exploit our more powerful database notation for a slightly different, but equivalent presentation.

Scheme 7.1 (Ideal and "Real" Encryption Systems) Let an encryption scheme $(\mathsf{gen_E}, \mathsf{E}, \mathsf{D}, \mathsf{pke\_len}, \mathsf{enc\_len})$ and parameters $n \in \mathbb{N}$ and s_{keys}, $s_{encs} : \mathbb{N} \Rightarrow \mathbb{N}$ be given. The functions s_{keys} and s_{encs} must be polynomially bounded in k. $s_{keys}(k)$ denotes the maximum number of keys generated in the system and $s_{encs}(k)$ the maximum number of encryptions per key. We define two systems

- $Sys^{\mathsf{enc,real}}_{n,s_{keys},s_{encs}} := \{(\{\mathsf{Enc}_{\mathcal{H}}\}, S_{\mathsf{enc},\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$,
- $Sys^{\mathsf{enc,sim}}_{n,s_{keys},s_{encs}} := \{(\{\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}\}, S_{\mathsf{enc},\mathcal{H}}) \mid \mathcal{H} \subseteq \{1, \ldots, n\}\}$.

For every $\mathcal{H}$, the ports are

- $Ports_{\mathsf{Enc}_{\mathcal{H}}} := Ports_{\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}} := \{\mathsf{in}_{\mathsf{enc},u}?, \mathsf{out}_{\mathsf{enc},u}!, \mathsf{out}_{\mathsf{enc},u}{}^{\triangleleft}! \mid u \in \mathcal{H}\}$,
- $S^c_{\mathsf{enc},\mathcal{H}} := \{\mathsf{in}_{\mathsf{enc},u}!, \mathsf{in}_{\mathsf{enc},u}{}^{\triangleleft}!, \mathsf{out}_{\mathsf{enc},u}? \mid u \in \mathcal{H}\}$.

Each machine maintains a key counter $curkey \in \mathbb{N}$ initialized with 0, and initially empty databases *keys* and *ciphers*. Entries in *keys* have attributes $(owner, skenc, pkenc, ec)$. Entries in *ciphers* have attributes $(msg, pkenc, ciph)$ and are used to look up intended cleartexts in the ideal system. The transition functions are given by the following rules. Let $\mathsf{in}_{\mathsf{enc},u}?$ be the port where the current input is made; the resulting output goes to $\mathsf{out}_{\mathsf{enc},u}!$ with $\mathsf{out}_{\mathsf{enc},u}{}^{\triangleleft}! := 1$.

- $pk \leftarrow (\mathsf{generate})$. (In $\mathsf{Enc}_{\mathcal{H}}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.) If $curkey < s_{keys}(k)$ then set $curkey{+}{+}$, $(sk, pk) \leftarrow \mathsf{gen_E}(1^k)$, and $keys :\Leftarrow (u, sk, pk, 0)$, where 0 is the initial value of an encryption counter, and return pk. Otherwise return $\downarrow$.
- $c \leftarrow (\mathsf{encrypt}, pk, m)$, for $pk, m \in \{0,1\}^+$. Let $K := keys[pkenc = pk]$. If $K = \downarrow$ or $K.ec \geq s_{encs}$, return $\downarrow$, else set $K.ec{+}{+}$ and
  - in $\mathsf{Enc}_{\mathcal{H}}$: Return $c \leftarrow \mathsf{E}_{pk}(m)$;

  – in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$: Set $c \leftarrow \mathsf{E}_{pk}(1^{|m|})$ and $ciphers :\Leftarrow (m, pk, c)$ and return c.

- $m \leftarrow (\mathsf{decrypt}, pk, c)$, for $pk, c \in \{0,1\}^+$. (Note that pk is used to designate the desired secret key.) Let $K := keys[pkenc = pk \wedge owner = u]$. If $K = \downarrow$, return $\downarrow$, else let $sk := K.skenc$ and
  – in $\mathsf{Enc}_{\mathcal{H}}$: Return $m := \mathsf{D}_{sk}(c)$;
  – in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$: Let $m := ciphers[pkenc = pk \wedge ciph = c].msg$. If $m \neq \downarrow$, return m, else return $m := \mathsf{D}_{sk}(c)$.[28]

$\diamond$

We need the following properties of these encryption machines.

Lemma 7.1 *The encryption systems have the following properties:*

1. *The two systems are computationally indistinguishable (without even a simulator), i.e.,* $Sys^{\mathsf{enc,real}}_{n,s_{keys},s_{encs}} \geq^{f,\mathsf{poly}}_{\mathsf{sec}} Sys^{\mathsf{enc,sim}}_{n,s_{keys},s_{encs}}$ *holds for the canonical mapping* f *and all parameters* $n \in \mathbb{N}$ *and* s_{keys}*,* $s_{encs} \in \mathbb{N}[x]$*.*
2. *Each transition is polynomial-time.*

$\square$

The first part was shown in [34] for *polynomials* s_{keys} and s_{encs}. Inspection of the proof easily reveals that this also holds for polynomially bounded functions. The second part is obvious.

7.1.2 Rewriting the Real System with Encryption Machines

We now rewrite the real machines M_u such that they use one of the encryption systems instead of doing all the work themselves. We call the modified machines M'_u. We set the parameters for the encryption system to $s_{keys} := s_{encs} := \mathsf{max\_in}(k) + 1$. This will ensure that the encryption system does not refuse actions that the overall system needs.

Each M'_u has three additional ports $\mathsf{in}_{\mathsf{enc},u}!$, $\mathsf{in}_{\mathsf{enc},u}{}^{\triangleleft}!$, and $\mathsf{out}_{\mathsf{enc},u}?$ for connecting to the encryption machine. It maintains a counter $steps_{\mathsf{out}_{\mathsf{enc},u}?}$ with bound $bound_{\mathsf{out}_{\mathsf{enc},u}?} := \mathsf{max\_in}(k) + 1$; the counter and length functions are treated as in the other systems. By "call the encryption machine with $y \leftarrow x$, expecting $y \ldots$" we abbreviate the same subroutine behavior as in the simulator, where M'_u uses the port $\mathsf{in}_{\mathsf{enc},u}!$.

The state is unmodified; only the actual entries in D_u for secret decryption keys are different.

The following transitions are modified:

- $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$.

  Set $sk^{\mathsf{hnd}} := curhnd_u$++; $pk^{\mathsf{hnd}} := curhnd_u$++. Call the encryption machine with $pk \leftarrow (\mathsf{generate})$, expecting $|pk| = \mathsf{pke\_len}(k)$, and store $D_u :\Leftarrow (sk^{\mathsf{hnd}}, \epsilon, \mathsf{ske}, ())$ and $D_u :\Leftarrow (pk^{\mathsf{hnd}}, (\mathsf{pke}, pk), \mathsf{pke}, ())$. Note that the secret key is only stored in the encryption machine.

[28] If two entries with this condition existed, the notation in [34] would lead to a non-deterministic choice, while we get $\downarrow$. However, the proof in [34] immediately yields that m in all such entries would be equal, and our definition of databases as sets joins equal entries.

- $c^* \leftarrow \mathsf{make\_enc}(pk^*, l)$, for $pk^*, l \in \{0,1\}^+$. (Note that this function is no longer state-less.)

  If $\mathsf{enc\_len}^*(k, |l|) > \mathsf{max\_len}(k)$, return $\downarrow$. Otherwise let $pk := pk^*[2]$, $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, and $l^* := (r, l)$. Call the encryption machine with $c \leftarrow (\mathsf{encrypt}, pk, l^*)$, expecting $|c| \leq \mathsf{max\_len}(k)$. If $c = \downarrow$ (it was an adversary key), encrypt directly as $c \leftarrow \mathsf{E}_{pk}(l^*)$. Set $c^* := (\mathsf{enc}, pk, c, r)$.

- $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  Parse c^{hnd} yielding $arg =: (pk^*)$. Return $\downarrow$ if $D_u[sk^{\mathsf{hnd}}].type \neq \mathsf{ske}$ or $D_u[c^{\mathsf{hnd}}].type \neq \mathsf{enc}$ or $D_u[sk^{\mathsf{hnd}}+1].word \neq pk^*$. Otherwise, set $pk := pk^*[2]$, $c := D_u[c^{\mathsf{hnd}}].word[3]$, and $r := D_u[c^{\mathsf{hnd}}].word[4]$. Call the encryption machine with $l^* \leftarrow (\mathsf{decrypt}, pk, c)$, expecting $|l^*| \leq |c|$. If $l^* = \downarrow$ or $l^*[1] \neq r$ or $l := l^*[2]$ is not a tagged list, then $l^{\mathsf{hnd}} := \downarrow$; else let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

The commands make_enc_keypair and parse_decrypt can be omitted, while parse_pke, encrypt, parse_enc, and pk_of_enc remain unchanged.

The following properties of the rewritten system are useful:

Lemma 7.2 The rewritten real machines M'_u have the following properties:

1. They are polynomial-time.
2. If they are used together with $\mathsf{Enc}_{\mathcal{H}}$ or $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$, no counter $steps_{\mathsf{out}_{\mathsf{enc},u}?}$, $curkey$, or $K.ec$ reaches its bound.
3. If used with $\mathsf{Enc}_{\mathcal{H}}$, their behavior is perfectly indistinguishable (for A and H) from the real system.

□

Proof.

1. Each machine M'_u is polynomial-time because the new input port $\mathsf{out}_{\mathsf{enc},u}!$ only allows a polynomial number of inputs because of the corresponding counter, the length function for this port is polynomially bounded and each modified transition only takes a polynomial number of steps.
2. First we consider a counter $steps_{\mathsf{out}_{\mathsf{enc},u}?}$ for $u \in \mathcal{H}$. Each output by $\mathsf{Enc}_{\mathcal{H}}$ or $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ at $\mathsf{out}_{\mathsf{enc},u}!$ is preceded by an input at $\mathsf{in}_{\mathsf{enc},u}!$, and M'_u only makes such an input upon a basic command, i.e., an input at $\mathsf{in}_u?$. Thus there are at most $\mathsf{max\_in}(k)$ such outputs.

   Similarly, the counter $curkey$ in $\mathsf{Enc}_{\mathcal{H}}$ or $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ is only incremented upon key generation commands input at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$, and thus at most $n\mathsf{max\_in}(k) < s_{keys}$ times. Each counter $keys.ec$ is only incremented upon encryption commands input at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$, and thus at most $n\mathsf{max\_in}(k) < s_{encs}$ times.
3. To prove equal outside behavior, we tacitly use Part 2. The major change is that secret decryption keys are no longer stored in D_u, but in *ciphers*. However, considering key generation and decryption together one easily sees that the same secret key is always used in both systems. As the search in the subroutine $:\leftarrow$ does not include secret keys, no other command operates on secret keys. The other change is that M'_u returns $\downarrow$ in make_enc if $|c| > \mathsf{max\_len}(k)$, while M_u returns c^*. However, this does not matter since M_u then rejects c^* in the surrounding command encrypt.

■

7.1.3 Replacing the Encryption Machine in the Real System

We now want to replace the system $\mathsf{Enc}_{\mathcal{H}}$ with $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$. This can be achieved by using the composition theorem of [34], i.e., it is possible to securely replace a subsystem of a larger system if the subsystem is as secure as the replacement in the sense of simulatability. We only have to show that the preconditions of the theorem are fulfilled. This is straightforward except for showing that the system on top of the replaced subsystem is polynomial-time, and this was shown in Lemma 7.2.

7.2 Combined System

We have to show that the modified real system, using the encryption machine, is at least as secure as the ideal system. We even show the stronger statement that the joint view of H and A, together with the buffers of connections leading to or from them, is indistinguishable.

In other words, we compare the joint machine $\mathsf{THSim}_{\mathcal{H}}$ consisting of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ with the combination $\mathsf{M}_{\mathcal{H}}$ of the correct machines M'_u with $u \in \mathcal{H}$ and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ (recall Figure 5).

A method to compare the observable behavior of two distributed systems is to define a mapping between their states and to show that the same input in corresponding states leads to the same output and corresponding states again, i.e., a bisimulation. However, the states of $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$ are not immediately comparable: a simulated state has no real versions for data that the adversary has not yet seen, while a real state has no global indices, adversary handles, and entries for data that are still unparsed in the correct machines. Similarly, the respective encryption machines are not always in a comparable state, e.g., because basic commands from the honest users cause the subsystems of $\mathsf{M}_{\mathcal{H}}$ to generate keys and ciphertexts, whereas for the subsystems of the simulator, this only happens when such a message is sent to A.

Therefore we define an intermediate system $\mathsf{C}_{\mathcal{H}}$ ("combined") with a combined state space of $\mathsf{M}_{\mathcal{H}}$ and $\mathsf{THSim}_{\mathcal{H}}$. We then compare $\mathsf{C}_{\mathcal{H}}$ with $\mathsf{M}_{\mathcal{H}}$ using a mapping from the states of $\mathsf{C}_{\mathcal{H}}$ to those of $\mathsf{M}_{\mathcal{H}}$, and similarly with $\mathsf{THSim}_{\mathcal{H}}$. By the transitivity of indistinguishability (of the families of views of the same A and H in all three configurations), we obtain the desired result.

7.2.1 Timing

The goal of a bisimulation is to prove that each input from A or H leads to the same outputs to A and H and to mapped states in the two pairs of configurations (except for the error sets). In this section, we show how this corresponds to our scheduling model.

We have already informally introduced machines $\mathsf{THSim}_{\mathcal{H}}$, $\mathsf{M}_{\mathcal{H}}$, and $\mathsf{C}_{\mathcal{H}}$ as combinations of other machines. However, a combination in the standard sense (formally defined in [34]) leaves all buffers separate, and thus its transitions are micro-transitions where only one submachine switches. Here we want macro-transitions where the submachines run until control is returned to A or H. Thus buffers that are entirely within the combination (i.e., their in, out and clock port) become a normal part of the state of the combined machine. Only for these macro-transitions can we hope for the bisimulation properties. We have to show that they are well-defined for our combinations.

$\mathsf{M}_{\mathcal{H}}$ is mainly a standard combination of the correct machines M'_u because A schedules all the network connections. Only the connections to $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ become internal. As at most one subroutine call to $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ is made in each transition, every macro-transition clearly terminates.

In $\mathsf{THSim}_{\mathcal{H}}$, the buffers in_{a} and out_{a} become internal. When $\mathsf{TH}_{\mathcal{H}}$ obtains a send command or $\mathsf{Sim}_{\mathcal{H}}$ a network input, the entire sequence of steps including id2real or real2id, respectively, becomes one macro-transition. Each one is a legitimate state transitions, because it terminates, as shown in Lemma 6.2, and leads to at most one output at each external port. (A send command leads to at most one network output, but possibly into duplicated buffers, and a network input leads to at most one user output.)

7.2.2 Definition of the Combined System

We now briefly define the combined system. Possible ambiguities will disappear below, where we compare the effects of all inputs in the three systems.

States of $\mathsf{C}_{\mathcal{H}}$ The main part of $\mathsf{C}_{\mathcal{H}}$ is a database D^* structured like D in $\mathsf{TH}_{\mathcal{H}}$. An entry x may have the following additional attributes:

- $x.word \in \{0,1\}^*$ is always defined and contains real data as in $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{Sim}_{\mathcal{H}}$ under the same handle(s). For $x.type \in \{\mathsf{sks}, \mathsf{ske}\}$, it is ϵ for adversary keys, i.e., if $\mathsf{owner}(x) = \mathsf{a}$ (recall Lemma 4.1.6), else a real secret key. For all other types, the word is non-empty.
- $x.parsed_u \in \{\mathsf{true}, \mathsf{false}\}$ for $u \in \mathcal{H}$ is $\downarrow$ if $x.hnd_u = \downarrow$; otherwise true indicates that the entry would be parsed in D_u, and false that it would still be of type null.
- $x.owner$ for ciphertexts with honest-user keys is adv if the ciphertext was received from the adversary, otherwise honest. For other ciphertexts it is $\downarrow$.
- $x.ec$ for secret encryption keys corresponds to the encryption counter in *ciphers* of the encryption machines.

Its state also contains variables *size* and $curhnd_u$ as in $\mathsf{TH}_{\mathcal{H}}$, the inner buffer variables out_{a} and in_{a}, all variables $steps_{\mathsf{p}?}$ as in $\mathsf{THSim}_{\mathcal{H}}$, and variables $steps_{\mathsf{out}_{\mathsf{enc},u}?}$ equal to the step counters in $\mathsf{M}_{\mathcal{H}}$.

Transitions of $\mathsf{C}_{\mathcal{H}}$ The D-part of $\mathsf{C}_{\mathcal{H}}$'s database D^*, the variables *size* and $curhnd_u$, and the ideal secure channels are treated exactly as in $\mathsf{TH}_{\mathcal{H}}$. An entry whose first handle $x.hnd_u$ is for $u \in \mathcal{H}$ gets the word that M'_u would contain under this handle except that also the secret keys from $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ are entered, and otherwise that from $\mathsf{Sim}_{\mathcal{H}}$. Thus, essentially, entries created due to basic commands from H get the words that M'_u and $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ would construct, while words received in network inputs from A are parsed completely and entered as by $\mathsf{Sim}_{\mathcal{H}}$. Outputs to H are made as in $\mathsf{TH}_{\mathcal{H}}$, outputs to A as in $\mathsf{M}_{\mathcal{H}}$.

7.2.3 Derivations

We now define the derivations of the original systems from the combined system. They are the mappings that we will show to be bisimulations. We use the following additional notation:

- Let ω abbreviate word lookup, i.e., $\omega(i) := D^*[i].word$ if $i \in \mathcal{HNDS}$, else $\omega(i) := i$. Let ω^*, applied to a list, denote that ω is applied to each element.
- We give most derived variables and entire machine states a superscript *, because in the bisimulation we have to compare them with the "original" versions. We make an exception with some variables of $\mathsf{THSim}_{\mathcal{H}}$ that are equal by construction in $\mathsf{C}_{\mathcal{H}}$; in particular D^* is $\mathsf{C}_{\mathcal{H}}$'s extended database and the derived D-part for $\mathsf{TH}_{\mathcal{H}}$ is immediately called D again.

We now assume that a state of $\mathsf{C}_{\mathcal{H}}$ is given and define derived states corresponding to the original systems.

$\mathsf{TH}_{\mathcal{H}}$**:** D**:** This is the restriction of D^* to all attributes except *word* and $parsed_u$.

$curhnd_u$ (for $u \in \mathcal{H} \cup \{\mathsf{a}\}$), *size*, and $steps_{\mathsf{p}?}$: All these variables are equal to those in $\mathsf{C}_{\mathcal{H}}$.

$\mathsf{M}_{\mathcal{H}}{}^*$**:** D_u^***:** (For every $u \in \mathcal{H}$.) We derive D_u^* as follows, starting with an empty database: For every $x^{\mathsf{hnd}} \le curhnd_u$, let $x := D^*[hnd_u = x^{\mathsf{hnd}}].ind$, $type := D^*[x].type$, and $m := D^*[x].word$. Then

- If $D^*[x].parsed_u = \mathsf{false}$, then $D_u^* :\Leftarrow (x^{\mathsf{hnd}}, m, \mathsf{null}, ())$.
- Else if $type \ne \{\mathsf{sks}, \mathsf{ske}\}$, then $D_u^* :\Leftarrow (x^{\mathsf{hnd}}, m, type, ())$.
- Else if $type = \mathsf{sks}$, then $D_u^* :\Leftarrow (x^{\mathsf{hnd}}, m, \mathsf{sks}, D^*[x].arg)$. (Both have a signature counter.)
- Else if $type = \mathsf{ske}$, then $D_u^* :\Leftarrow (x^{\mathsf{hnd}}, \epsilon, \mathsf{ske}, ())$. (A secret key of an honest user is only in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.)

$curhnd_u^*$**:** This variable equals $curhnd_u$ of $\mathsf{C}_{\mathcal{H}}$.

$steps_{\mathsf{p}?}^*$**:** These variables equal $steps_{\mathsf{p}?}$ of $\mathsf{C}_{\mathcal{H}}$, except for $\mathsf{p}? = \mathsf{net}_{u,v,x}?$ with $(u,v,x) \in ch\_honest$, where they equal $steps_{\mathsf{net\_id}_{u,v,x}?}$.

$net_{u,v,x}^*$**:** (For every $(u,v,x) \in ch\_honest$.) Let $net_{u,v,x}^* := \omega^*(net\_id_{u,v,x})$.

$\mathsf{Sim}_{\mathcal{H}}{}^*$**:** D_{a}^***:** We derive D_{a}^* as follows, starting with an empty database: For all $x^{\mathsf{hnd}} \le curhnd_{\mathsf{a}}$, let $x := D^*[hnd_{\mathsf{a}} = x^{\mathsf{hnd}}].ind$, $type := D^*[x].type$, and $m := D^*[x].word$.

- If $type \notin \{\mathsf{pks}, \mathsf{pke}\}$ (i.e., not a public key), then $D_{\mathsf{a}}^* :\Leftarrow (x^{\mathsf{hnd}}, m, ())$.
- If $type \in \{\mathsf{pks}, \mathsf{pke}\}$, let $sk^{\mathsf{ind}} := x - 1$. If $\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a}$, then $D_{\mathsf{a}}^* :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{adv}))$, else $D_{\mathsf{a}}^* :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{honest}, \omega(sk^{\mathsf{ind}})))$.

$curhnd_{\mathsf{a}}^*$**:** This variable equals $curhnd_{\mathsf{a}}$ of $\mathsf{C}_{\mathcal{H}}$.

$steps_{\mathsf{p}?}$**:** These variables are identical to those in $\mathsf{C}_{\mathcal{H}}$.

$\mathsf{THSim}_{\mathcal{H}}$**:** $out_{\mathsf{a}}, in_{\mathsf{a}}$**:** These buffer contents are equal to those in $\mathsf{C}_{\mathcal{H}}$, and we claim that they are empty after each macro-transition.

$net\_id_{u,v,x}$**:** (For every $(u,v,x) \in ch\_honest$.) The buffers of messages on secure channels are equal to those in $\mathsf{C}_{\mathcal{H}}$.

$\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}^*$**:** $keys^*$**:** We derive $keys^*$ as follows, starting with an empty database: For all $sk^{\mathsf{ind}} \le size$ with $D^*[sk^{\mathsf{ind}}].type = \mathsf{ske}$ and $u := \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$, let $keys^* :\Leftarrow (u, D^*[sk^{\mathsf{ind}}].word[2], D^*[sk^{\mathsf{ind}} + 1].word[2], D^*[sk^{\mathsf{ind}}].ec)$. Thus we take the owner, untagged secret key and counter from the secret-key entry and the untagged public key from the corresponding public-key entry.

$ciphers^*$**:** We derive $ciphers^*$ as follows, starting with an empty database: For all $c^{\mathsf{ind}} \le size$ with $D^*[c^{\mathsf{ind}}].type = \mathsf{enc}$ and $D^*[c^{\mathsf{ind}}].owner = \mathsf{honest}$, let $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}) := D^*[c^{\mathsf{ind}}].arg$, $(l, pk^*, c^*) := \omega^*((l^{\mathsf{ind}}, pk^{\mathsf{ind}}, c^{\mathsf{ind}}))$, and $l^* = (c^*[4], l)$. Then set $ciphers^* :\Leftarrow (l^*, pk^*[2], c^*[3])$.

$curkey^*$**:** Let $curkey^* = |keys^*|$.

The buffers to and from $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}^*$ are empty after each macro-transition.

In the comparison between $\mathsf{C}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$ and $\mathsf{THSim}_{\mathcal{H}}$, we show that the derived states and outputs are the same as in the original systems except for certain cases that we collect in (families of) *error sets*. We show in Section 7.7 that the error sets have negligible probability.

7.2.4 Invariants in $\mathsf{C}_{\mathcal{H}}$

For the bisimulation, we need invariants about $\mathsf{C}_{\mathcal{H}}$. Those that concern only the D-part of the state have already been presented in Lemma 4.1. They are *index and handle uniqueness*, *well-defined terms*, *message correctness*, *key secrecy*, *length bounds*, and *correct key pairs*. This lemma also holds for $\mathsf{C}_{\mathcal{H}}$ because $\mathsf{C}_{\mathcal{H}}$ always treats the D-part of its state with transitions of $\mathsf{TH}_{\mathcal{H}}$. We now present new invariants.

- *Fully defined.* For every $x \in D^*$, the attributes $x.ind$, $x.type$, $x.arg$, $x.length$, and $x.word$ are never $\downarrow$.
- *Word uniqueness.* For each word $m \in \{0,1\}^*$, we have $|D^*[word = m \wedge type \notin \{\mathsf{sks}, \mathsf{ske}\}]| \leq 1$.
- *Correct length.* For all $i \leq size$, $D[i].len = |D^*[i].word|$, except if $D[i].type \in \{\mathsf{sks}, \mathsf{ske}\}$.
- *No unparsed secret keys.* If $u \in \mathsf{owners}(D^*[i]) \cap \mathcal{H}$ and $D^*[i].parsed_u = \mathsf{false}$, then $D^*[i].type \notin \{\mathsf{sks}, \mathsf{ske}\}$.
- *Correct arguments.* For all $i \leq size$, the real message $m := D^*[i].word$ and the abstract type and arguments, $type^{\mathsf{id}} := D^*[i].type$ and $arg^{\mathsf{ind}} := D^*[i].arg$, are compatible. More precisely, let $arg^{\mathsf{real}} := \omega^*(arg^{\mathsf{ind}})$. Then we require:
  - If $m = \epsilon$, then $type^{\mathsf{id}} \in \{\mathsf{sks}, \mathsf{ske}\}$.
  - If $type^{\mathsf{id}} \notin \{\mathsf{sks}, \mathsf{ske}\}$, let $(type, arg^{\mathsf{parse}}) := \mathsf{parse}(m)$. Then $type = type^{\mathsf{id}}$, and:
    * If $type \notin \{\mathsf{sig}, \mathsf{enc}\}$, then $arg^{\mathsf{parse}} = arg^{\mathsf{real}}$.
    * If $type = \mathsf{sig}$, then $arg^{\mathsf{parse}} = arg^{\mathsf{real}}[1,2]$. (Parsing does not output a counter.)
    * If $type = \mathsf{enc}$, then $arg^{\mathsf{parse}}[1] = arg^{\mathsf{real}}[1]$.
      Further, let $o := D^*[i].owner$ and $sk^{\mathsf{ind}} := arg^{\mathsf{ind}}[1] - 1$. Then $(\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a} \iff o = \downarrow)$. For $o = \mathsf{honest}$, we only claim that $l^{\mathsf{ind}} := arg^{\mathsf{ind}}[2] \neq \downarrow$.
      For $o = \mathsf{adv}$, we claim that the decrypted and looked-up cleartexts are equal: Let $sk^* := \omega(sk^{\mathsf{ind}})$, $l^{\mathsf{ind}} := arg^{\mathsf{ind}}[2]$, and $l := \mathsf{parse\_decrypt}(sk^*, m)$. We claim that $l = \omega(l^{\mathsf{ind}})$.
- *Strongly correct arguments if* $\mathsf{a} \notin \mathsf{owners}(D^*[i])$ *or* $D^*[i].owner = \mathsf{honest}$. Let $type := D^*[i].type$, $arg^{\mathsf{ind}} := D^*[i].arg$ and $arg^{\mathsf{real}} := \omega^*(arg^{\mathsf{ind}})$. Then $type \neq \mathsf{garbage}$ and $m := D^*[i].word$ has the following probability distribution:[29]
  - If $type \in \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}\}$, then $m \leftarrow \mathsf{make\_}type(arg^{\mathsf{real}})$.
  - If $type \in \{\mathsf{sks}, \mathsf{pks}, \mathsf{ske}, \mathsf{pke}\}$, the formula is applied to pairs, i.e., we have $pk^{\mathsf{ind}} = sk^{\mathsf{ind}} + 1$ with appropriate types, and $(sk^*, pk^*) \leftarrow \mathsf{make\_sig\_keypair}()$ or $(sk^*, pk^*) \leftarrow \mathsf{make\_enc\_keypair}()$, respectively, for $(sk^*, pk^*) := \omega^*(sk^{\mathsf{ind}}, pk^{\mathsf{ind}})$.
  - If $type = \mathsf{sig}$, then arg^{ind} is of the form $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c)$ with $c \in \mathbb{N}$ and $c \leq \mathsf{max\_skc}(k)$. Let $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$ and $arg'^{\mathsf{real}} := \omega^*(sk^{\mathsf{ind}}, l^{\mathsf{ind}}, c)$. Then $a \neq \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$ and $m \leftarrow \mathsf{make\_sig}(arg'^{\mathsf{real}})$.

[29] Here one sees that the bisimulation is probabilistic, i.e., we actually consider distributions of states before and after a transition. This invariant says that in such a state distribution, and given the mentioned arguments, m is distributed as described independent of other state parts.

- If $type = \mathsf{enc}$, then arg^{real} is of the form (pk^*, l) and m of the form (enc, pk, c, r) with $pk = pk^*[2]$ and where r is distributed as $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$. Further, $o := D^*[i].owner \neq \mathsf{adv}$.

  The distribution of c depends on o: If $o = \mathsf{honest}$ (a ciphertext among honest users), then $c \leftarrow \mathsf{E}_{pk}(1^{|(r,l)|})$. If $o = \downarrow$ (a ciphertext from an honest user to the adversary), then $c \leftarrow \mathsf{E}_{pk}((r,l))$.

- *Word secrecy.* We require that the adversary never obtains information about nonce-like word components without adversary handles. For this, we define a set *Pub_Var* of "public" variables that A may know or get to know later as follows: It contains

  - all words $D^*[i].word$ with $D^*[i].hnd_{\mathsf{a}} \neq \downarrow$;
  - the state of A and H;
  - the $\mathsf{TH}_{\mathcal{H}}$-part of the state of $\mathsf{C}_{\mathcal{H}}$ and the ideal secure channels;
  - secret keys where the public keys are public, i.e., if $D^*[i].hnd_{\mathsf{a}} \neq \downarrow$ and $D^*[i].type \in \{\mathsf{pks}, \mathsf{pks}\}$, then also $D^*[i-1].word$.[30]

  We claim that at all times, no information from outside has flown into *Pub_Var* in the sense of information flow in programming languages (static program analysis).

"Word secrecy" implies that no information from "official" nonces n', random values r in signatures and ciphertexts, and public keys has flown into *Pub_Var* unless these values have adversary handles. There, the treatment of absence of information flow in the static sense implies absence of Shannon information. We need this stronger notion for the bisimulation because it is inductive: If two variables x and y individually obtain no information flow about a variable z, then neither does the pair (x, y), in contrast to Shannon information.

The following definition summarizes what we plan to do with these invariants:

Definition 7.1 *(Bisimulation Property) By "an input retains all invariants" we mean the following:*

- *The resulting macro-transition of $\mathsf{C}_{\mathcal{H}}$ retains the invariants if they were true before the input.*
- *If the input is made to $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{THSim}_{\mathcal{H}}$ 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 $\mathsf{C}_{\mathcal{H}}$. We call this "correct derivation".*

All conditions are obviously true initially, when all databases and buffers are empty and the counters 0. ◇

We use the following general lemma about transitions of $\mathsf{C}_{\mathcal{H}}$:

Lemma 7.3 The combined machine $\mathsf{C}_{\mathcal{H}}$ has the following property:

1. The only modifications to existing entries x in D^* of $\mathsf{C}_{\mathcal{H}}$ are assignments to previously undefined attributes $x.hnd_u$ together with $x.parsed_u := \mathsf{false}$, changes of $x.parsed_u$ from false to true, and (via the commands sign and $\mathsf{encrypt}$) modifications to the counters in entries of type sks and ske.

[30]These secret keys are included because information from them flows into the public keys, signatures, and decryptions, but they do not get adversary handles when those values are published. Note that we do not use this analysis for security proofs about the signature and encryption systems.

2. The function owner of a secret key never changes.
3. If a state change in $\mathsf{C}_{\mathcal{H}}$ only consists in a new handle $x.hnd_u$ in an existing entry x in D^* and an assignment to $x.parsed_u$, only the following invariants have to be shown:
    - "Correct derivation" of D_u and $curhnd_u^*$.
    - "Word secrecy" if $u = \mathsf{a}$.

□

Proof. Part 1 follows immediately from the definition of $\mathsf{C}_{\mathcal{H}}$, and can easily be verified by inspection of the detailed commands.

For Part 3, it is clear that the other derivations are not affected. "Fully defined", "word uniqueness", and "correct length" are clear, and so is "correct arguments" given Part 2. "Strongly correct arguments" is required for fewer entries if an adversary handle is added, and clear for those. For "no unparsed secret keys", our preconditions never occur for secret keys by "key secrecy". For "word secrecy" in the case $u \neq \mathsf{a}$, the new handle does not change which words belong to Pub_Var, and the information in $x.hnd_u$ is from the D-part of D^* and thus within Pub_Var. ■

7.3 Comparison of Basic Commands

We first consider the effects of a basic command c input at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$. Recall that the actions of $\mathsf{C}_{\mathcal{H}}$ on a large part of its state are by definition equal to those of $\mathsf{TH}_{\mathcal{H}}$, and so is $\mathsf{C}_{\mathcal{H}}$'s output at $\mathsf{out}_u!$. We will not always mention this again.

7.3.1 General Aspects

Lemma 7.4 For an individual basic command c input at $\mathsf{in}_u?$, we can assume it is well-formed and we only have to show the following properties:

- The outputs at $\mathsf{out}_u!$ in $\mathsf{C}_{\mathcal{H}}$ (where it is defined to be as in $\mathsf{TH}_{\mathcal{H}}$) and $\mathsf{M}_{\mathcal{H}}$ are the same.
- "Correct derivation" of D_u and $curhnd_u$.
- For the commands $\mathsf{gen\_enc\_keypair}$ and $\mathsf{encrypt}$, "correct derivation" of the state of $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.
- The invariants in D^* are retained, where "word secrecy" is already clear.

□

Proof. The length functions in $\mathsf{TH}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$, and M'_u are defined from identical variables $steps_{\mathsf{in}_u?}$, with identical bounds $bounds_{\mathsf{in}_u?} = \mathsf{max\_in}(k)$, and from identical input domains. The latter follows because the domains are textually equal, and the only dynamic condition is that handle parameters are $< curhnd_u$, which is also equal. Hence $\mathsf{C}_{\mathcal{H}}$ and M'_u obtain the same actual inputs. Then they both increment $steps_{\mathsf{in}_u?}$. Next follow domain tests, which are equal as we just saw. Hence the general conventions lead to abortion in the same cases. Hence from now on, we assume that c is well-formed.

Now we exploit that c is local in both $\mathsf{TH}_{\mathcal{H}}$ and M'_u. First this implies that the only output in all systems is made at $\mathsf{out}_u!$, and that all buffer contents and remaining step variables remain correctly derived.

Secondly, the submachine $\mathsf{Sim}_{\mathcal{H}}$ of $\mathsf{THSim}_{\mathcal{H}}$ is not involved, nor is any other machine M'_v. We show that their derived counterparts D^*_v and $curhnd^*_v$ with $v \in \mathcal{H} \cup \{a\} \setminus \{u\}$ are also unchanged. Locality also means that $\mathsf{TH}_{\mathcal{H}}$ and thus $\mathsf{C}_{\mathcal{H}}$ does not modify handles for such a value v. Thus $curhnd^*_v$ is indeed unchanged, and the database D^*_v can change at most by changes to a non-handle attribute of an existing entry x of D^* with $x.hnd_v \neq \downarrow$. By Lemma 7.3.1, this can only happen if $x.type = \mathsf{sks}$ and in a sign command. (The attribute $x.ec$ for $x.type = \mathsf{ske}$ is only used in the derivation of $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.) However, by "key secrecy" (Lemma 4.1.6), this would imply $x.hnd_u = \downarrow$, and we easily see below that this is not the case in such a sign command (honest users only sign with their own keys). This proves all the derivations that are not shown individually.

Basic commands other than $\mathsf{gen\_enc\_keypair}$ and $\mathsf{encrypt}$ clearly do not change the state of $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$. Further, they do not modify entries of type ske, pke and enc, upon which the derivation relies.

Finally, we show word secrecy: The output at $\mathsf{out}_u!$ and the updates to the D-part of D^* are made entirely with commands of $\mathsf{TH}_{\mathcal{H}}$ and thus within Pub_Var. New or existing words only get a handle for u, so that nothing is added to Pub_Var. ■

7.3.2 One-level Parsing

Many command executions in M'_u contain a subroutine "parse x^{hnd} (if necessary)". We show that it retains the invariants and ensures $D_u[x^{\mathsf{hnd}}].type \neq \mathsf{null}$.

The subroutines assign $D_u[x^{\mathsf{hnd}}].type$ by parsing $D_u[x^{\mathsf{hnd}}].word = D^*[hnd_u = x^{\mathsf{hnd}}].word$ if it was still null. By "correct arguments" this is $D^*[hnd_u = x^{\mathsf{hnd}}].type$ unless we would use it for secret keys, which is excluded by "no unparsed secret keys". Here $\mathsf{C}_{\mathcal{H}}$ sets $D^*[hnd_u = x^{\mathsf{hnd}}].parsed_u := \mathsf{true}$ if it was still false. Thus in the derived D^*_u we change from $(x^{\mathsf{hnd}}, m, \mathsf{null}, ())$ to $(x^{\mathsf{hnd}}, m, D^*[hnd_u = x^{\mathsf{hnd}}].type, ())$. (Again, exceptions for secret keys are excluded by "no unparsed secret keys".) This retains "correct derivation" of D_u, the only invariant that could be affected, and ensures $D_u[x^{\mathsf{hnd}}].type \neq \mathsf{null}$.

7.3.3 Type and Length Queries

We only have to show equal outputs because no state changes are made (beyond possibly that in parsing).

- *Type query:* $t \leftarrow \mathsf{get\_type}(x^{\mathsf{hnd}})$.

  In $\mathsf{TH}_{\mathcal{H}}$ and thus in $\mathsf{C}_{\mathcal{H}}$, we get $t = D^*[hnd_u = x^{\mathsf{hnd}}].type$. In M'_u, parsing ensures $D_u[x^{\mathsf{hnd}}].type \neq \mathsf{null}$, and thus by "correct derivation" $t = D_u[x^{\mathsf{hnd}}].type = D^*[hnd_u = x^{\mathsf{hnd}}].type$. This is equal.

- *Length query:* $l \leftarrow \mathsf{get\_len}(x^{\mathsf{hnd}})$.

  In $\mathsf{TH}_{\mathcal{H}}$, we get $l = D^*[hnd_u = x^{\mathsf{hnd}}].len$. In M'_u, parsing ensures $D_u[x^{\mathsf{hnd}}].type \neq \mathsf{null}$. Thus, if $D_u[x^{\mathsf{hnd}}].type \not\in \{\mathsf{sks}, \mathsf{ske}\}$, we get $l = |D_u[x^{\mathsf{hnd}}].word| = |D^*[hnd_u = x^{\mathsf{hnd}}].word|$. By the invariant "correct length", this is equal. For $D_u[x^{\mathsf{hnd}}].type \in \{\mathsf{sks}, \mathsf{ske}\}$, both output $l = 0$.

7.3.4 Storing and Retrieving Data

- *Storing:* $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m)$, for $m \in \{0,1\}^*$.

  Let $m^* := (\mathsf{data}, m)$. Correct arguments for the type data means that for every i, $D^*[i].type = \mathsf{data}$ and $D^*[i].arg = (m)$ if and only if $D^*[i].word = m^*$. $\mathsf{TH}_{\mathcal{H}}$ first searches for the former kind of entry, M'_u for the latter in the derived D_u.

If it exists and has a u-handle m^{hnd}, both return m^{hnd}. (By word and handle uniqueness, there is at most one.)

If it exists but has no u-handle, $\mathsf{TH}_{\mathcal{H}}$ and thus $\mathsf{C}_{\mathcal{H}}$ assigns a new handle $m^{\mathsf{hnd}} := \mathit{curhnd}$++. M'_u verifies (in the subroutine $:\leftarrow$) that $|m^*| \leq \mathsf{max\_len}(k)$; this is true by "correct length" for $D^*[i]$. Thus it makes a new entry $(m^{\mathsf{hnd}}, m^*, \mathsf{data}, ())$ with the same handle (by "correct derivation" of curhnd_u). Thus $\mathsf{C}_{\mathcal{H}}$ sets $D^*[i].\mathit{parsed}_u := \mathsf{true}$. The outputs are equal and "correct derivation" is retained. By Lemma 7.3.3, this is all we have to show.

If there is no such entry, $\mathsf{TH}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$, verifies $\mathsf{data\_len}^*(|m|) \leq \mathsf{max\_len}(k)$. M'_u acts as in the previous case. The length tests are equivalent because $\mathsf{data\_len}^*(|m|) = \mathsf{list\_len}(|\mathsf{data}|, |m|) = |m^*|$. Then both make an entry with $m^{\mathsf{hnd}} := \mathit{curhnd}$++ and resulting in $D^* :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{data}, \mathit{arg} := (m), \mathit{hnd}_u := m^{\mathsf{hnd}}, \mathit{len} := \mathsf{data\_len}^*(|m|), \mathit{parsed}_u := \mathsf{true}, \mathit{word} := m^*)$. The outputs are equal, and "correct derivation" is clearly retained. The invariants are shown as follows: "Fully defined" and "no unparsed secret keys" are obvious. "Word uniqueness" holds by the preconditions of this case. "Correct length" holds because we already showed $|m^*| = \mathsf{data\_len}^*(|m|)$. "Correct arguments" was already written out above. "Strongly correct arguments" holds by construction.

- *Retrieval:* $m \leftarrow \mathsf{retrieve}(m^{\mathsf{hnd}})$.

  Let $m^{\mathsf{ind}} := D^*[\mathit{hnd}_u = m^{\mathsf{hnd}}].\mathit{ind}$ and $m := D^*[m^{\mathsf{ind}}].\mathit{word}$. $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if $D^*[m^{\mathsf{ind}}].\mathit{type} \neq \mathsf{data}$, and M'_u if $D_u[m^{\mathsf{hnd}}] \neq \mathsf{data}$ after parsing. This is equivalent. Otherwise $\mathsf{TH}_{\mathcal{H}}$ returns $D^*[m^{\mathsf{ind}}].\mathit{arg}[1]$, and M'_u returns $\mathsf{parse\_data}(D_u[m^{\mathsf{hnd}}].\mathit{word})[1]$. This is equal by "correct derivation" and "correct arguments".

7.3.5 Lists

- *Generate a list:* $l^{\mathsf{hnd}} \leftarrow \mathsf{list}(x_1^{\mathsf{hnd}}, \ldots, x_j^{\mathsf{hnd}})$, for $j \geq 0$.

  Let $x_i := D^*[\mathit{hnd}_u = x_i^{\mathsf{hnd}}].\mathit{ind}$ for $i = 1, \ldots, j$. $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if any $D^*[x_i].\mathit{type} \in \{\mathsf{sks}, \mathsf{ske}\}$, and M'_u if $D_u[x_i^{\mathsf{hnd}}].\mathit{type} \in \{\mathsf{sks}, \mathsf{ske}\}$. This is equivalent even though $D_u[x_i^{\mathsf{hnd}}].\mathit{type}$ may still be null, i.e., $D^*[x_i].\mathit{parsed}_u = \mathsf{false}$, by "no unparsed secret keys". Otherwise let $l := (\mathsf{list}, D^*[x_1].\mathit{word}, \ldots, D^*[x_j].\mathit{word})$.

  The rest is similar to store; thus we are briefer: By "correct arguments", for every i we have $D^*[i].\mathit{type} = \mathsf{list}$ and $D^*[i].\mathit{arg} = (x_1, \ldots, x_j)$ iff $D^*[i].\mathit{word} = l$. If such an entry exists, both $\mathsf{TH}_{\mathcal{H}}$ and M'_u determine and return a handle for u to it; M'_u's verification that $|l| \leq \mathsf{max\_len}(k)$ (in $:\leftarrow$) is true by "correct length" for $D^*[i]$. Thus they make the same output and retain "correct derivation".

  If no such entry exists, $\mathsf{TH}_{\mathcal{H}}$ sets $\mathit{length} := \mathsf{list\_len}^*(D^*[x_1].\mathit{len}, \ldots, D^*[x_j].\mathit{len})$ and verifies $\mathit{length} \leq \mathsf{max\_len}(k)$. M'_u acts as in the previous case. The length tests are equivalent because $\mathit{length} = \mathsf{list\_len}(|\mathsf{list}|, D^*[x_1].\mathit{len}, \ldots, D^*[x_j].\mathit{len}) = |l|$ by "correct length" for the entries $D^*[x_i]$, which are not secret keys. Then both make an entry with $l^{\mathsf{hnd}} := \mathit{curhnd}$++ and resulting in $D^* :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{list}, \mathit{arg} := (x_1, \ldots, x_j), \mathit{hnd}_u := l^{\mathsf{hnd}}, \mathit{len} := \mathit{length}, \mathit{parsed}_u := \mathsf{true}, \mathit{word} := l)$.

  The outputs are equal and "correct derivation" is clearly retained. The invariants "fully defined" and "no unparsed secret keys" are obvious, "word uniqueness" holds by the preconditions of this case, "correct length" was already shown, and "correct arguments" and "strongly correct arguments" are obvious.

- *i-th projection:* $x^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, i)$, for $i \in \mathbb{N}$.

  Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[hnd_u = l^{\mathsf{hnd}}].type \neq \mathsf{list}$. This is equivalent because M'_u first parses. Otherwise $\mathsf{TH}_{\mathcal{H}}$ uses $x_i^{\mathsf{ind}} := D^*[hnd_u = l^{\mathsf{hnd}}].arg[i]$, while M'_u parses $D^*[hnd_u = l^{\mathsf{hnd}}].word$, yielding arg, and sets $x_i := arg[i]$. By "correct arguments", $D^*[x_i^{\mathsf{ind}}].word = x_i$.

  If $x^{\mathsf{hnd}} := D^*[x_i^{\mathsf{ind}}].hnd_u$ already exists, $\mathsf{TH}_{\mathcal{H}}$ returns it, and as $D^*[x_i^{\mathsf{ind}}].type \notin \{\mathsf{sks}, \mathsf{ske}\}$ by "key secrecy", M'_u also finds this entry in the subroutine $:\leftarrow$ and returns x^{hnd}. Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds it as $x^{\mathsf{hnd}} := curhnd_u$++. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word x_i, and thus makes a new entry $(x^{\mathsf{hnd}}, x_i, \mathsf{null}, ())$ with the same handle. (Its test $|x_i| \leq \mathsf{max\_len}(k)$ is true by "correct length" for $D^*[x_i^{\mathsf{ind}}]$.) Equal outputs and "correct derivation" is clear. By Lemma 7.3.3, this is all we have to show.

7.3.6 Nonces

- *Generation:* $x^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$.

  Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u set $x^{\mathsf{hnd}} := curhnd_u$++ and make a new entry. In $\mathsf{C}_{\mathcal{H}}$ this results in $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{nonce}, arg := (), hnd_u := n^{\mathsf{hnd}}, len := \mathsf{nonce\_len}^*(k), parsed_u := \mathsf{true}, word := (\mathsf{nonce}, n'))$ with $n' \stackrel{\mathcal{R}}{\leftarrow} \{0,1\}^{\mathsf{nonce\_len}(k)}$.

  Equal outputs and "correct derivation" are clear, and so are "fully defined" and "no unparsed secret keys". If "word uniqueness" is not fulfilled, i.e., n' equals an old nonce, we put the run in an error set *Nonce_Coll*. "Correct length" is fulfilled by obvious computation, and "correct arguments" and "strongly correct arguments" obviously.

7.3.7 Signatures

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_sig\_keypair}()$.

  Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u set $sk^{\mathsf{hnd}} := curhnd_u$++; $pk^{\mathsf{hnd}} := curhnd_u$++ and make two new entries. In $\mathsf{C}_{\mathcal{H}}$ this gives $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{sks}, arg := (0), hnd_u := sk^{\mathsf{hnd}}, len := 0, parsed_u := \mathsf{true}, word := sk^*)$ and $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{pks}, arg := (), hnd_u := pk^{\mathsf{hnd}}, len := \mathsf{pks\_len}^*(k), parsed_u := \mathsf{true}, word := pk^*)$, where $(sk^*, pk^*) \leftarrow \mathsf{make\_sig\_keypair}()$.

  The outputs are equal. "Correct derivation" follows with the special definition for type sks. "Fully defined" is clear, and "no unparsed secret keys" holds because the secret key is parsed. If "word uniqueness" is not fulfilled, we put the run in an error set *Key_Coll*. In this case, pk matches an already existing value. (Recall that "word uniqueness" is not required for entries of type sks.) "Correct length" is fulfilled by the exception for secret keys, and because $\mathsf{pks\_len}^*(k) = \mathsf{list\_len}(|\mathsf{pks}|, \mathsf{pks\_len}'(k)) = |pk^*|$ by definition of $\mathsf{make\_sig\_keypair}$. Under "correct arguments", nothing is required for type sks, and for pks it is clearly fulfilled. "Strongly correct arguments" is obvious.

- *Signature generation:* $s^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $sk^{\mathsf{ind}} := D^*[hnd_u = sk^{\mathsf{hnd}}].ind$ and $l^{\mathsf{ind}} := D^*[hnd_u = l^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[sk^{\mathsf{ind}}].type \neq \mathsf{sks}$ or $D^*[l^{\mathsf{ind}}].type \neq \mathsf{list}$ or $D^*[sk^{\mathsf{ind}}].arg[1] \geq \mathsf{max\_skc}(k)$. Their tests are equivalent by "correct derivation" and although M'_u only ensures that $D_u[l^{\mathsf{hnd}}]$ is parsed by "no unparsed secret keys".

  Further, $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if $length := \mathsf{sig\_len}^*(k, D^*[l^{\mathsf{ind}}].len) > \mathsf{max\_len}(k)$. Else it sets $s^{\mathsf{hnd}} := curhnd_u$++, $pk^{\mathsf{ind}} := sk^{\mathsf{ind}} + 1$ and $c := D^*[sk^{\mathsf{ind}}].arg[1]$++.

Then $\mathsf{TH}_{\mathcal{H}}$ makes a new entry $D :\Leftarrow (ind := size\text{++}, type := \mathsf{sig}, arg := (pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c), hnd_u := s^{\mathsf{hnd}}, len := length)$.

M'_u uses $(sk^*, l) := \omega(sk^{\mathsf{ind}}, l^{\mathsf{ind}})$ and sets $c := D_u[sk^{\mathsf{hnd}}].add\_arg[1]\text{++}$ and $s^* \leftarrow \mathsf{make\_sig}(sk^*, l, c)$. If $|s^*| > \mathsf{max\_len}(k)$, it decrements $D_u[sk^{\mathsf{hnd}}].add\_arg[1]$ again and returns $\downarrow$. This length test equals that in $\mathsf{TH}_{\mathcal{H}}$: By "strongly correct arguments", the key pair including sk^* was generated with $\mathsf{make\_sig\_keypair}$ (because by "key secrecy" the secret key has at most one handle, and this is $D^*[sk^{\mathsf{ind}}].hnd_u$ by the input condition). With the notation from inside $\mathsf{make\_sig}$, this means that sk was correctly generated, and thus by Section 5.1.1, we have $|sig| = \mathsf{sig\_len}(k, \mathsf{max\_skc}(k), |(r, l)|) = \mathsf{sig\_len}'(k, |l|)$. This yields $|s^*| = \mathsf{sig\_len}^*(k, |l|)$, and by "correct length" for the entry $D^*[l^{\mathsf{ind}}]$ (which is not a secret key) this is what $\mathsf{TH}_{\mathcal{H}}$ verified.

Hence either both do not change their state and return $\downarrow$, or both make the described updates and M'_u sets $s^{\mathsf{hnd}} := curhnd_u\text{++}$ and makes an entry $D_u :\Leftarrow (s^{\mathsf{hnd}}, s^*, \mathsf{sig}, ())$.

The outputs are equal, and the update to $D^*[sk^{\mathsf{ind}}]$ retains "correct derivation" because the counters are updated consistently, and no invariants is affected.

Now we consider the new signature entry: "Correct derivation" is clear if we augment $\mathsf{TH}_{\mathcal{H}}$'s entry with the word s^* and $parsed_u = \mathsf{true}$. "Fully defined" and "no unparsed secret keys" are clear. If "word uniqueness" is not fulfilled, then r within s^* equals an old value in the same place in a word; hence we put the run in an error set $Nonce\_Coll$. "Correct length" is fulfilled as shown above.

"Correct arguments" follows by comparing the output format of $\mathsf{make\_sig}$ with the predicate in $\mathsf{parse\_sig}$. Here we exploit that correctly generated signatures for correct key pairs are always valid (Section 5.1.1), that l is a tagged list by "correct arguments" for $D^*[l^{\mathsf{ind}}]$, and that $D^*[pk^{\mathsf{ind}}] = pk^*$ as returned by parsing by "strongly correct arguments" for the key pair. "Strongly correct arguments" holds by construction and because $c \leq \mathsf{max\_skc}(k)$.

- *Signature verification:* $v \leftarrow \mathsf{verify}(s^{\mathsf{hnd}}, pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $s^{\mathsf{ind}} := D^*[hnd_u = s^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[s^{\mathsf{ind}}].type \neq \mathsf{sig}$. (Indeed M'_u has parsed the entry.) Otherwise, let $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c) := D^*[s^{\mathsf{ind}}].arg$, $s^* := D^*[s^{\mathsf{ind}}].word$, and $(pk^*, l) \leftarrow \mathsf{parse\_sig}(s^*)$. By "correct arguments" for the entry $D^*[s^{\mathsf{ind}}]$, we have $(pk^*, l) = \omega^*(pk^{\mathsf{ind}}, l^{\mathsf{ind}})$.

  Then $\mathsf{TH}_{\mathcal{H}}$ outputs true if $pk^{\mathsf{hnd}} = D^*[pk^{\mathsf{ind}}].hnd_u$ and $l^{\mathsf{hnd}} = D^*[l^{\mathsf{ind}}].hnd_u$, and M'_u if $D^*[hnd_u = pk^{\mathsf{hnd}}].word = pk^*$ and $D^*[hnd_u = l^{\mathsf{hnd}}].word = l$. By the previous paragraph, the former clearly implies the latter. By "word uniqueness", the latter also implies the former. Otherwise both output false.

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_sig}(s^{\mathsf{hnd}})$.

  We start exactly as in signature verification: Let $s^{\mathsf{ind}} := D^*[hnd_u = s^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[s^{\mathsf{ind}}].type \neq \mathsf{sig}$. (Indeed M'_u has parsed the entry.) Otherwise, let $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c) := D^*[s^{\mathsf{ind}}].arg$, $s^* := D^*[s^{\mathsf{ind}}].word$, and $(pk^*, l) \leftarrow \mathsf{parse\_sig}(s^*)$. By "correct arguments" for the entry $D^*[s^{\mathsf{ind}}]$, we have $(pk^*, l) = \omega^*(pk^{\mathsf{ind}}, l^{\mathsf{ind}})$.

  The rest is as in list projection: If $D^*[pk^{\mathsf{ind}}].hnd_u$ already exists, both return it. Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds it as $pk^{\mathsf{hnd}} := curhnd_u\text{++}$. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word pk^*, and thus makes a new entry $(pk^{\mathsf{hnd}}, pk^*, \mathsf{null}, ())$ with the same handle. (Its test $|pk^*| \leq \mathsf{max\_len}(k)$ is true by "correct length" for $D^*[pk^{\mathsf{ind}}]$.) Equal outputs and "correct derivation" are clear. By Lemma 7.3.3, this is all we have to show.

- *Message retrieval:* $l^{\mathsf{hnd}} \leftarrow \mathsf{msg\_of\_sig}(s^{\mathsf{hnd}})$.

  This is exactly like public-key retrieval, only with the second argument, l^{ind}, of $D^*[s].arg$.

7.3.8 Public-key Encryption

For the first two of the following commands, Lemma 7.4 requires us to also consider the encryption machine. Recall that the counters *curkey* and $K.ec$ of entries in *keys* never reach their bounds. (This follows from Lemma 7.2.2 inductively with the fact that the derived real system that we actually consider equals a normal real one.)

- *Key generation:* $(sk^{\mathsf{hnd}}, pk^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$.

  Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u set $sk^{\mathsf{hnd}} := curhnd_u$++, $pk^{\mathsf{hnd}} := curhnd_u$++.

  Then M'_u calls the encryption machine with $pk \leftarrow (\mathsf{generate})$ at port $\mathsf{in}_{\mathsf{enc},u}?$. As *curkey* has not reached its bound, $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ generates a key pair as $(sk, pk) \leftarrow \mathsf{gen}_{\mathsf{E}}(1^k)$ and makes an entry $(u, sk, pk, 0)$ in *keys*. This implies $|pk| = \mathsf{pke\_len}(k)$ and thus M'_u accepts pk.

  Then $\mathsf{TH}_{\mathcal{H}}$ and M'_u each make two new entries. In $\mathsf{C}_{\mathcal{H}}$, this gives $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{ske}, arg := (), hnd_u := sk^{\mathsf{hnd}}, len := 0, parsed_u := \mathsf{true}, word := (\mathsf{ske}, sk), ec := 0)$ and $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{pke}, arg := (), hnd_u := pk^{\mathsf{hnd}}, len := \mathsf{pke\_len}^*(k), parsed_u := \mathsf{true}, word := (\mathsf{pke}, pk))$.

  Equal outputs and "correct derivation" of D_u and $curhnd_u$ are clear. In the derived *keys**, the new secret-key entry in D^* leads to an entry $(u, sk, pk, 0)$. This equals the new entry in *keys*. The counters *curkey* and *curkey** clearly remain equal, and nothing changes in *ciphers* and *ciphers**.

  "Fully defined" is clear, and "no unparsed secret keys" holds because the secret key is parsed. "Word uniqueness" is only required for the public key. If it is not fulfilled, and thus pk equals an old key, we put the run in an error set *Key_Coll*. "Correct length" is only needed for the public key and follows from the definition of $\mathsf{pke\_len}^*$. "Correct arguments" is only needed for the public key and clear there. "Strongly correct arguments" holds by simple comparison with the algorithm $\mathsf{make\_enc\_keypair}$.

- *Encryption:* $c^{\mathsf{hnd}} \leftarrow \mathsf{encrypt}(pk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $pk^{\mathsf{ind}} := D^*[hnd_u = pk^{\mathsf{hnd}}].ind$ and $l^{\mathsf{ind}} := D^*[hnd_u = l^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[pk^{\mathsf{ind}}].type \neq \mathsf{pke}$ or $D^*[l^{\mathsf{ind}}].type \neq \mathsf{list}$. (Indeed, M'_u has parsed both entries.) Let $(pk^*, l) := \omega^*((pk^{\mathsf{ind}}, l^{\mathsf{ind}}))$ and $pk := pk^*[2]$. We have $|pk| = \mathsf{pke\_len}(k)$ even if it is an adversary key because "correct arguments" for $D^*[pk^{\mathsf{ind}}]$ implies that $\mathsf{parse\_pke}(pk^*)$ does not return $\downarrow$.

  Further, $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if $length := \mathsf{enc\_len}^*(k, D^*[l^{\mathsf{ind}}].len) > \mathsf{max\_len}(k)$, and so does M'_u if $\mathsf{enc\_len}^*(k, |l|) > \mathsf{max\_len}(k)$ (by the rewritten algorithm $\mathsf{make\_enc}$). This is equivalent by "correct length" for $D^*[l^{\mathsf{ind}}]$.

  Now M'_u sets $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$ and $l^* := (r, l)$, and calls the encryption machine with $c \leftarrow (\mathsf{encrypt}, pk, l^*)$ at port $\mathsf{in}_{\mathsf{enc},u}?$. We distinguish two cases, where $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$ and $v := \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$:

  - If $v \in \mathcal{H}$, then by "correct derivation" of *keys*, there is an entry K in *keys* with $K.pkenc = pk$. $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ sets $K.ec$++, and $\mathsf{C}_{\mathcal{H}}$ reflects that in $D^*[sk^{\mathsf{ind}}].ec$++. Then $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ sets $c \leftarrow \mathsf{E}_{pk}(1^{|l^*|})$ and $ciphers :\Leftarrow (l^*, pk, c)$ and returns c. Clearly $|c| = \mathsf{enc\_len}'(k, |l|)$.

– If $v = \mathsf{a}$, then by "correct derivation" of *keys* and "word uniqueness", there is no such entry K in *keys*. Then $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ immediately returns $\downarrow$, and M'_u itself encrypts $c \leftarrow \mathsf{E}_{pk}(l^*)$. Again we have $|c| = \mathsf{enc\_len}'(k, |l|)$. (Recall that pk is of correct length even for adversary keys.)

In both cases, M'_u sets $c^* := (\mathsf{enc}, pk, c, r)$, and we get $|c^*| = \mathsf{enc\_len}^*(k, |l|)$. Thus c^* passes the length test in the outer algorithm $\mathsf{encrypt}$.

Now both $\mathsf{TH}_\mathcal{H}$ and M'_u set $c^{\mathsf{hnd}} := curhnd_u{+}{+}$ and make a new entry. In $\mathsf{C}_\mathcal{H}$, this gives $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{enc}, arg := (pk^{\mathsf{ind}}, l^{\mathsf{ind}}), hnd_u := c^{\mathsf{hnd}}, len := length, parsed_u := \mathsf{true}, word := c^*)$, and additionally $D^*[size].owner := \mathsf{honest}$ if $v \in \mathcal{H}$.

The outputs are equal. "Correct derivation" of D_u and $curhnd_u$ is clear for the new entry, and the counter update in $D^*[sk^{\mathsf{ind}}]$ does not influence this derivation. If $v = \mathsf{a}$, then $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ does not change its state and the new entry in D^* does not influence $\mathsf{Enc}^*_{\mathsf{sim},\mathcal{H}}$ either. If $v \in \mathcal{H}$, the update $K.ec{+}{+}$ occurs in both *keys* and *keys**. Further, the new entry in D^* leads to an entry $((c^*[4], l), pk^*[2], c^*[3])$ in *ciphers**, which equals the new entry in *ciphers*.

Among the invariants, "fully defined" and "no unparsed secret keys" are clear. If "word uniqueness" is not fulfilled, then r equals a previous value in an encryption entry. Hence we put the run in an error set *Nonce_Coll*. "Correct length" was already shown. For "correct arguments", only the public-key equality and the conditions involving o are required in both our cases, which are obvious. "Strongly correct arguments" holds by construction, exactly according to our two cases.

- *Decryption:* $l^{\mathsf{hnd}} \leftarrow \mathsf{decrypt}(sk^{\mathsf{hnd}}, c^{\mathsf{hnd}})$.

  Let $sk^{\mathsf{ind}} := D^*[hnd_u = sk^{\mathsf{hnd}}].ind$ and $c^{\mathsf{ind}} := D^*[hnd_u = c^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_\mathcal{H}$ and M'_u return $\downarrow$ if $D^*[sk^{\mathsf{ind}}].type \neq \mathsf{ske}$ or $D^*[c^{\mathsf{ind}}].type \neq \mathsf{enc}$. Their tests are equivalent by "correct derivation" and although M'_u only ensures that $D_u[c^{\mathsf{hnd}}]$ is parsed by "no unparsed secret keys". Otherwise, let $pk^{\mathsf{ind}} := D^*[c^{\mathsf{ind}}].arg[1]$, $c^* := D^*[c^{\mathsf{ind}}].word$, and $(pk^*) \leftarrow \mathsf{parse\_enc}(c^*)$. By "correct arguments" for the entry $D^*[c^{\mathsf{ind}}]$, we have $pk^* = D^*[pk^{\mathsf{ind}}].word$.

  Next, $\mathsf{TH}_\mathcal{H}$ returns $\downarrow$ if $pk^{\mathsf{ind}} \neq sk^{\mathsf{ind}} + 1$, while M'_u sets $pk^{\mathsf{hnd}} := sk^{\mathsf{hnd}} + 1$ and returns $\downarrow$ if $D_u[pk^{\mathsf{hnd}}].word \neq pk^*$. This is equivalent to $\mathsf{TH}_\mathcal{H}$'s test: We have $u = \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$ by key secrecy and because there is the handle sk^{hnd} for u. Hence if $pk^{\mathsf{ind}} = sk^{\mathsf{ind}} + 1$, then by "correct key pairs" we have $D^*[pk^{\mathsf{ind}}].hnd_u = pk^{\mathsf{hnd}}$ and the word is indeed pk^*. Else by "word uniqueness" it is not.

  Now both machines are willing to decrypt. $\mathsf{TH}_\mathcal{H}$ uses $l^{\mathsf{ind}} := D^*[c^{\mathsf{ind}}].arg[2]$. We have $l^{\mathsf{ind}} = \downarrow$ or $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$, as $D^*[c^{\mathsf{ind}}]$ must have been constructed with the command $\mathsf{encrypt}$ or $\mathsf{adv\_invalid\_ciph}$. Hence $l := \omega(l^{\mathsf{ind}})$ is either $\downarrow$ or, by "correct arguments", a tagged list. We show that M'_u's internal result equals l.

  M'_u sets $pk := pk^*[2]$ and $(c, r) := c^*[3, 4]$ and calls the encryption machine with $l^* \leftarrow (\mathsf{decrypt}, pk, c)$ at port $\mathsf{in}_{\mathsf{enc},u}!$. By "correct derivation" of *keys*, there is an entry $K = (u, sk, pk, ec) \in keys$, where $sk = sk^*[2]$ for $sk^* := D^*[sk^{\mathsf{ind}}].word$. $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ finds exactly this entry because *pkenc* is a key attribute. Now it decrypts to some l^*. We distinguish two cases according to $o := D^*[c^{\mathsf{ind}}].owner$, where $o \neq \downarrow$ because $u = \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$.

  – If $o = \mathsf{honest}$, then by "correct arguments", $l^{\mathsf{ind}} \neq \downarrow$. Thus by "correct derivation", *ciphers* contains the entry $e := ((r, l), pk, c)$. If there is another entry e' with the same pk and c and another attribute $e'.msg$, we put the run in an error set $Ciph\_Coll$. Thus we now assume that $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ finds entry e. Hence it returns (r, l), which passes M'_u's tests in $\mathsf{decrypt}$. Hence M'_u obtains l.

– If $o = \mathsf{adv}$, no entry in $ciphers$ is derived from $D^*[c^{\mathsf{ind}}]$. Typically $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ will therefore return $l^* := \mathsf{D}_{sk}(c)$. Together with the rest of $\mathsf{decrypt}$ this means that M'_u derives its result as $l' := \mathsf{parse\_decrypt}(sk^*, c^*)$. (The additional length test $|l^*| \le |c|$ is fulfilled by Section 5.1.2.) "Correct arguments" for this case implies $l' = l$.

– If $o = \mathsf{adv}$, there might nevertheless be an entry $((r_1, l_1), pk, c) \in ciphers$, derived from another entry in D^* with $D^*[c_1^{\mathsf{hnd}}].owner = \mathsf{honest}$ and (using "word uniqueness"), $D^*[c_1^{\mathsf{hnd}}].arg[1] = pk^{\mathsf{ind}}$, $\omega(D^*[c_1^{\mathsf{hnd}}].arg[2]) = l_1$, and $D^*[c_1^{\mathsf{hnd}}].word = (\mathsf{enc}, pk, c, r_1)$. "Word uniqueness" implies $c_1^* \neq c^*$ and thus $r_1 \neq r$. Then $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ returns (r_1, l_1) and in $\mathsf{decrypt}$, the comparison with r from c^* fails. (This is the non-malleability of the encryption system with additional randomization.) Hence M'_u sets $l := \downarrow$.

  We have to show that also $l^{\mathsf{ind}} = \downarrow$. "Strongly correct arguments" applies to $D^*[c_1^{\mathsf{hnd}}]$. Thus c was chosen as $\mathsf{E}_{pk}(\mathsf{1}^{|(r_1,l_1)|})$. This implies $\mathsf{D}_{sk}(c) = \mathsf{1}^{|(r_1,l_1)|}$. Hence for the entry $D^*[c^{\mathsf{ind}}]$, where "correct arguments" applies, we have $\mathsf{parse\_decrypt}(sk^*, c^*) = \downarrow$ because the intermediate result $l^* = \mathsf{1}^{|(r_1,l_1)|}$ is not a list. This implies $l^{\mathsf{ind}} = \downarrow$ as desired.

The next part is as in list projection: If $D^*[l^{\mathsf{ind}}].hnd_u$ already exists, both return it. Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds it as $l^{\mathsf{hnd}} := curhnd_u$++. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word l, and thus makes a new entry $(l^{\mathsf{hnd}}, l, \mathsf{list}, ())$ with the same handle. (Its test $|l| \le \mathsf{max\_len}(k)$ is true by "correct length" for $D^*[l^{\mathsf{ind}}]$.) Equal outputs are clear, and for "correct derivation" $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$ was already shown. By Lemma 7.3.3, this is all we have to show.

- *Public-key retrieval:* $pk^{\mathsf{hnd}} \leftarrow \mathsf{pk\_of\_enc}(c^{\mathsf{hnd}})$.

  This works almost exactly as for signatures: Let $c^{\mathsf{ind}} := D^*[hnd_u = c^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and M'_u return $\downarrow$ if $D^*[c^{\mathsf{ind}}].type \neq \mathsf{enc}$. (Indeed M'_u has parsed the entry.) Otherwise, let $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}) := D^*[c^{\mathsf{ind}}].arg$, $c^* := D^*[c^{\mathsf{ind}}].word$, and $(pk^*) \leftarrow \mathsf{parse\_enc}(c^*)$. By "correct arguments" for the entry $D^*[c^{\mathsf{ind}}]$, we have $pk^* = \omega(pk^{\mathsf{ind}})$.

  If $D^*[pk^{\mathsf{ind}}].hnd_u$ already exists, both return it. Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds it as $pk^{\mathsf{hnd}} := curhnd_u$++. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word pk^*, and thus makes a new entry $(pk^{\mathsf{hnd}}, pk^*, \mathsf{null}, ())$ with the same handle. (Its test $|pk^*| \le \mathsf{max\_len}(k)$ is true by "correct length" for $D^*[pk^{\mathsf{ind}}]$.) Equal outputs and "correct derivation" are clear. By Lemma 7.3.3, this is all we have to show.

7.4 Send Command from Honest User

We now consider an input $\mathsf{send}\_x(v, l_u^{\mathsf{hnd}})$ at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$. Intuitively, this part of the proof shows that the adversary does not get any information in the real system that it cannot get in the ideal system, because any real information can be simulated indistinguishably given only the outputs from $\mathsf{TH}_{\mathcal{H}}$.

7.4.1 General Aspects

As with basic commands, the length function for the port $\mathsf{in}_u?$, and thus the actual input, is equal in $\mathsf{TH}_\mathcal{H}$, and thus $\mathsf{C}_\mathcal{H}$, and M'_u. Then they all increment $steps_{\mathsf{in}_u?}$ and make the same domain test. Hence from now on, we assume that the input is well-formed. Let $l^{\mathsf{ind}} := D^*[hnd_u = l_u^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_\mathcal{H}$ and M'_u continue only if $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$. (The conditions are equivalent by "correct derivation" and because M'_u has parsed the entry.)

Now M'_u always outputs $l := D^*[l^{\mathsf{ind}}].word$ (here we used "correct derivation" again) at port $\mathsf{net}_{u,v,x}!$ without further action. We distinguish two overlapping cases.

Secure Channel: $(u, v, x) \in ch\_honest$. Then $\mathsf{TH}_\mathcal{H}$ and $\mathsf{C}_\mathcal{H}$ output l^{ind} at $\mathsf{net\_id}_{u,v,x}!$ without further action. The only invariant that could be affected is correct derivation of secure-channel buffers. This holds by definition for $\mathsf{THSim}_\mathcal{H}$. For $\mathsf{M}_\mathcal{H}$, we retain $net^*_{u,v,x} = \omega^*(net\_id_{u,v,x})$ because we already showed $l = D^*[l^{\mathsf{ind}}].word$.

Output to Adversary: $(u, v, x) \in ch\_to\_adv$. Here $\mathsf{net}_{u,v,x}!$ is connected to A, and thus $\mathsf{C}_\mathcal{H}$ outputs l there like M'_u at the end. However, it first continues to update the $\mathsf{TH}_\mathcal{H}$-part of its state.

If $l^{\mathsf{hnd}} := D^*[l^{\mathsf{ind}}].hnd_\mathsf{a}$ already exists, $\mathsf{TH}_\mathcal{H}$ outputs $(u, v, x, l^{\mathsf{hnd}})$ at $\mathsf{out_a}!$ and schedules it. By Lemma 6.2.5, handles output by $\mathsf{TH}_\mathcal{H}$ are always accepted 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 also holds for the counters in $\mathsf{C}_\mathcal{H}$; this follows (inductively) from the equality of these values with the state of $\mathsf{THSim}_\mathcal{H}$.

Thus $\mathsf{Sim}_\mathcal{H}$ accepts the resulting input. By "correct derivation", $\mathsf{Sim}_\mathcal{H}$ finds $D_\mathsf{a}[l^{\mathsf{hnd}}].word = D^*[l^{\mathsf{ind}}].word = l$ and outputs it at $\mathsf{net}_{u,v,x}!$. This ensures equal outputs. Also "correct derivation" of the buffers out_a and in_a and the corresponding counters is clear. The only other invariant that could be affected is "word secrecy". It follows because the output already had an a-handle, so that the flow is within Pub_Var.

Thus the case $D^*[l^{\mathsf{ind}}].hnd_\mathsf{a} = \downarrow$ remains. Here $\mathsf{TH}_\mathcal{H}$ assigns $l^{\mathsf{hnd}} := curhnd_\mathsf{a}\mathsf{++}$ and outputs $(u, v, x, l^{\mathsf{hnd}})$ at $\mathsf{out_a}!$. As above, this input is accepted by $\mathsf{Sim}_\mathcal{H}$. By "correct derivation", $\mathsf{Sim}_\mathcal{H}$ finds $D_\mathsf{a}[l^{\mathsf{hnd}}] = \downarrow$. It sets $curhnd_\mathsf{a}\mathsf{++}$, thus maintaining "correct derivation" of these variables. Then $\mathsf{Sim}_\mathcal{H}$ calls $l \leftarrow \mathsf{id2real}(l^{\mathsf{hnd}})$ and outputs l at $\mathsf{net}_{u,v,x}!$. It is therefore sufficient to show that $\mathsf{id2real}$ together with the commands it inputs to $\mathsf{TH}_\mathcal{H}$ retains all invariants and yields $l = D^*[l^{\mathsf{ind}}].word$.

7.4.2 General Aspects of id2real

We use an inductive proof that id2real retains all invariants and produces the right output. The following lemma defines the weaker conditions that hold for all subcalls, and proves some invariants once and for all.

Lemma 7.5 It is sufficient to show the following for each recursive call $m \leftarrow \mathsf{id2real}(m^{\mathsf{hnd}})$:

- Initially,
  - $m^{\mathsf{hnd}} \leq curhnd_\mathsf{a}$,
  - the entry $e := D^*[hnd_\mathsf{a} = m^{\mathsf{hnd}}]$ has $e.type = \mathsf{list}$ or is a component of another entry, and
  - the entry e already existed with $\mathsf{a} \notin \mathsf{owners}(e)$.

- The following invariants hold before every subcall and at the end:
  - *Weakly correct derivation of* D_{a}*:* We have $curhnd_{\mathsf{a}} = curhnd^*_{\mathsf{a}}$ and D_{a} is a subset of D^*_{a}.
  - *Weak word secrecy.* Secret information only must not flow into the smaller set Pub_Var' containing only the words $D_{\mathsf{a}}[i^{\mathsf{hnd}}].word$ instead of all $D^*[i^{\mathsf{ind}}].word$ with $D^*[i^{\mathsf{ind}}].hnd_{\mathsf{a}} \neq \downarrow$, and the corresponding secret keys.
- At the end, the following holds:
  - *Result:* The result is $m = m^* := e.word$.
  - *Progress:* If D_{a} and $curhnd_{\mathsf{a}}$ are the values before the call, and D'_{a} and $curhnd'_{\mathsf{a}}$ those after the call, then D'_{a} is a superset of D_{a} and contains entries $D'_{\mathsf{a}}[m^{\mathsf{hnd}}]$ and $D'_{\mathsf{a}}[x^{\mathsf{hnd}}]$ for all $curhnd_{\mathsf{a}} < x \leq curhnd'_{\mathsf{a}}$.

The initial conditions and invariants are fulfilled before the outermost call. □

Proof. All normal invariants were true before $\mathsf{C}_{\mathcal{H}}$, like $\mathsf{TH}_{\mathcal{H}}$, assigned $D^*[l^{\mathsf{ind}}].hnd_{\mathsf{a}} := l^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{+}{+}$ before the outermost call. After this assignment, they still hold except that the entry $D^*_{\mathsf{a}}[l^{\mathsf{hnd}}]$ is missing in D_{a}, and that we do not know about "word secrecy" for the entry with the new adversary handle. Thus the weaker invariants of this lemma hold. The initial conditions also hold because $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$ was explicitly verified, and the entry had $D^*[l^{\mathsf{ind}}].hnd_{\mathsf{a}} = \downarrow$ before the assignment.

As only $D^*_{\mathsf{a}}[l^{\mathsf{hnd}}]$ is missing before the outermost call, "weakly correct derivation" and "progress" imply that normal "correct derivation" of D_{a} holds after this call. This also implies "word secrecy". Furthermore, the result is then $l = D^*[hnd_{\mathsf{a}} = l^{\mathsf{hnd}}].word = D^*[l^{\mathsf{ind}}].word$, as required.

We now show why the remaining invariants hold automatically. "Correct derivation" of the state of $\mathsf{TH}_{\mathcal{H}}$ holds by definition, as well as of the buffers out_{a} and in_{a} and the corresponding counters. These buffers are empty after the macro-transition by the subroutine behavior within id2real.

In $\mathsf{M}_{\mathcal{H}}$ and its encryption machine $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$, nothing changes, and nor does it in the derived version because only the command adv_parse is executed in $\mathsf{TH}_{\mathcal{H}}$ during id2real, and it only assigns new adversary handles (recall Lemma 7.3.3).

As $\mathsf{C}_{\mathcal{H}}$ only assigns adversary handles in D^*, the invariants except for "word secrecy" are unaffected. ■

Now we consider one of these recursive calls $m \leftarrow \mathsf{id2real}(m^{\mathsf{hnd}})$ and show that it retains the invariants of Lemma 7.5. Let $m^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].ind$ and $m^* := D^*[m^{\mathsf{ind}}].word$ and $(m^{\mathsf{ind}}_1, \ldots, m^{\mathsf{ind}}_j) := D^*[m^{\mathsf{ind}}].arg$.

7.4.3 Steps 1 and 2: Parsing and Handle Updates

In Step 1, id2real calls $(type, (m^{\mathsf{hnd}}_1, \ldots, m^{\mathsf{hnd}}_j)) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. By definition, this gives $type := D^*[m^{\mathsf{ind}}].type$, and, except if $type = \mathsf{enc}$, also $(m^{\mathsf{hnd}}_1, \ldots, m^{\mathsf{hnd}}_j) := \mathsf{ind2hnd}^*_{\mathsf{a}}((m^{\mathsf{ind}}_1, \ldots, m^{\mathsf{ind}}_j))$.

We show that the domain expectations of id2real are fulfilled:

- $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$: For garbage this follows from "strongly correct arguments", because $D^*[m^{\mathsf{ind}}]$ already existed without adversary handle by the preconditions of Lemma 7.5. For sks, ske it follows from key secrecy, because $D^*[m^{\mathsf{ind}}]$ is of type list or a component of another entry.

- $j \leq \mathsf{max\_len}(k)$: The only type of entry in D^* with multiple arguments is list, and in list generation $\mathsf{TH}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$, verifies this condition.

- $m_i^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ if $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ follows from Lemma 6.2.5. (We already showed that it also holds in $\mathsf{C}_{\mathcal{H}}$.)

- Otherwise $|m_i^{\mathsf{hnd}}| \leq \mathsf{max\_len}(k)$ holds because arguments that are not handles only occur for the following types: data, where $\mathsf{TH}_{\mathcal{H}}$ verified it when storing; sks, which was excluded; sig, where it is a counter c that can become at most $\mathsf{max\_skc}(k)$ by "strongly correct arguments", so that $|c| \leq \mathsf{max\_len}(k)$ follows from the conditions on length functions; enc if the ciphertext length len is output; then it follows from Lemma 4.1.4.

The only resulting change in D^* is that new a-handles may have been assigned to some of the entries $D^*[m_i^{\mathsf{ind}}]$. (We do not mention the step-counters because we already know "correct derivation" for them and that they do not reach their bounds.) Clearly these handles are in $\mathcal{HNDS}$ and appear in increasing order in $(m_1^{\mathsf{hnd}}, \ldots, m_j^{\mathsf{hnd}})$.

Hence Step 2 of $\mathsf{id2real}$ updates $curhnd_{\mathsf{a}}$ consistently. This reestablishes "weak correct derivation of D_{a}". Further, it clearly retains "weak word secrecy".

7.4.4 Step 3: Recursion

In Step 3, $\mathsf{id2real}$ considers the values $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$. For those with $D_{\mathsf{a}}[m_i^{\mathsf{hnd}}] \neq \downarrow$, it sets $m_i := D_{\mathsf{a}}[m_i^{\mathsf{hnd}}].word$ without any state changes, so that the invariants are clearly retained. For the others, it makes recursive calls $m_i \leftarrow \mathsf{id2real}(m_i^{\mathsf{hnd}})$.

We have to show that the initial conditions from Lemma 7.5 hold for these calls.

- $m_i^{\mathsf{hnd}} \leq curhnd_{\mathsf{a}}$ is clear by the updates of $curhnd_{\mathsf{a}}$ in Step 2.

- The entries $D^*[hnd_{\mathsf{a}} = m_i^{\mathsf{hnd}}]$ are components of the entry $D^*[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}]$ by definition of components and $\mathsf{adv\_parse}$.

- The entry $D^*[hnd_{\mathsf{a}} = m_i^{\mathsf{hnd}}]$ already existed before the outermost call (because no new entries are made). At that time $\mathsf{a} \notin \mathsf{owners}(D^*[hnd_{\mathsf{a}} = m_i^{\mathsf{hnd}}])$ must have been true because normal "correct derivation of D_{a}" held.

Thus by the induction hypothesis, these calls retain the invariants.

We show that they also establish the second "progress" property for our call $\mathsf{id2real}(m^{\mathsf{hnd}})$: The first "progress" property of each subcall, the existence of an entry $D_{\mathsf{a}}[m_i^{\mathsf{hnd}}]$, means that entries in D_{a} now exist for all values assigned to $curhnd_{\mathsf{a}}$ directly in our call $\mathsf{id2real}(m^{\mathsf{hnd}})$ (in Step 2). The second "progress" property of the subcalls ensures that entries also exist for all further increments they made to $curhnd_{\mathsf{a}}$. Hence indeed entries exist for all new values of $curhnd_{\mathsf{a}}$ of our call and its subcalls.

By the "result" properties of the subcalls, we obtain $m_i = D^*[hnd_{\mathsf{a}} = m_i^{\mathsf{hnd}}].word = D^*[m_i^{\mathsf{ind}}].word = \omega(m_i^{\mathsf{ind}})$ for all $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$. As $\mathsf{id2real}$ sets $m_i := m_i^{\mathsf{hnd}} = m_i^{\mathsf{ind}}$ for the other i's, it obtains

$$arg^{\mathsf{real}} := (m_1, \ldots, m_j) = \omega^*(arg^{\mathsf{ind}}),$$

except in certain cases for $type = \mathsf{enc}$, and where arg^{real} equals that in the formulation of "strongly correct arguments".

7.4.5 Step 4: General Aspects of Type-specific Part

Now $\mathsf{id2real}$ proceeds depending on *type*. Each of these variants ends with an assignment to m, which is then output, and $D_\mathsf{a} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$ for certain arguments add_arg. No further changes to D_a are made and $\mathsf{TH}_\mathcal{H}$ is not involved.

Lemma 7.6 For the individual types, we only have to show:

- a correct result $m = m^*$, where we can assume "strongly correct arguments" for m^*, i.e., the entry $D^*[m^{\mathsf{ind}}]$;
- "correct derivation" of add_arg in the new entry, and only for $type \in \{\mathsf{pks}, \mathsf{pke}\}$;
- "word secrecy" for m, i.e., no flow of secret information into m, where arguments m_i are not secret information.

□

Proof. "Strongly correct arguments" holds for $D^*[m^{\mathsf{ind}}]$ because it had no adversary handle initially by Lemma 7.5. The rest of "weakly correct derivation" follows from the correctness of m and because add_arg is always () for the other types, as required (recall that the types sks and ske are not possible). The new entry in D_a gives us the missing first part of "progress". For "weak word secrecy", only m is added to Pub_Var. All values m_i already belong to Pub_Var: If $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ they were entered into D_a at the latest in Step 3, and if $m_i^{\mathsf{hnd}} \notin \mathcal{HNDS}$ they were output by $\mathsf{TH}_\mathcal{H}$. ■

7.4.6 Data, Lists, and Nonces

If $type \in \{\mathsf{data}, \mathsf{list}, \mathsf{nonce}\}$, then $\mathsf{id2real}$ sets $m \leftarrow \mathsf{make\_}type(arg^{\mathsf{real}})$.

"Strongly correct arguments" for these types means that m^* also has the probability distribution $m^* \leftarrow \mathsf{make\_}type(arg^{\mathsf{real}})$. Thus derivation of D_a^* gives the same distribution as we get in D_a.

"Word secrecy" for m holds for data and list because their generation depends only on the arguments m_i (the list elements or the raw data). For nonces it holds since each nonce generation does not depend on prior information.

7.4.7 Public Signature Keys

If $type = \mathsf{pks}$, then $\mathsf{id2real}$ sets $(sk^*, pk^*) \leftarrow \mathsf{make\_sig\_keypair}()$, $m := pk^*$ and $add\_arg := (\mathsf{honest}, sk^*)$.

Let $sk^{\mathsf{ind}} = m^{\mathsf{ind}} - 1$ and $(sk^{*\mathsf{real}}, pk^{*\mathsf{real}}) := (D^*[sk^{\mathsf{ind}}].word, m^*)$. By "strongly correct arguments" $(sk^{*\mathsf{real}}, pk^{*\mathsf{real}})$ was chosen with $\mathsf{make\_sig\_keypair}()$. "Correct key pairs" implies $\mathsf{a} \neq \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$, because otherwise $D^*[m^{\mathsf{ind}}]$ would also have got an a-handle at once. In the derived D_a^*, we therefore have an entry $(m^{\mathsf{hnd}}, pk^{*\mathsf{real}}, (\mathsf{honest}, sk^{*\mathsf{real}}))$ with the same distribution as $\mathsf{id2real}$'s choice.

"Word secrecy" for $m = pk^*$ holds since key generation does not depend on prior information, and both pk^* and sk^* become elements of Pub_Var.

7.4.8 Signatures

If $type = \mathsf{sig}$, "strongly correct arguments" implies that arg^{ind} is of the form $(pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c)$ with $c \in \mathbb{N}$. Let $(pk^*, l, c) := arg^{\mathsf{real}} = \omega^*(arg^{\mathsf{ind}})$. First this implies $c \neq \mathsf{false}$, as claimed in $\mathsf{id2real}$. Let $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$ and $sk^* := \omega(sk^{\mathsf{ind}})$. By "strongly correct arguments", m^* is distributed as $m^* \leftarrow \mathsf{make\_sig}(sk^*, l, c)$. Furthermore, it implies $a \neq \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$. By "correct derivation" of D_{a} for the existing entries, this implies $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk^*)$, where $pk^{\mathsf{hnd}} = D^*[pk^{\mathsf{ind}}]$. This proves the second format claim in id2real, and id2real sets $m \leftarrow \mathsf{make\_sig}(sk^*, l, c)$. This is the same distribution.

For proving "word secrecy" for m, we only have to consider the parameter sk^*, because l and c are parameters m_i (and make_sig is functional). By definition of "word secrecy", sk^* belongs to Pub_Var, but may contain information from the corresponding public key pk^*. As this is our parameter m_1, it belongs to Pub_Var already. This proves "word secrecy".

7.4.9 Public Encryption Keys

If $type = \mathsf{pke}$, then id2real sets $(sk^*, pk^*) \leftarrow \mathsf{make\_enc\_keypair}()$, $m := pk^*$, and $add\_arg := (\mathsf{honest}, sk^*)$.

Let $sk^{\mathsf{ind}} = m^{\mathsf{ind}} - 1$ and $(sk^{*\mathsf{real}}, pk^{*\mathsf{real}}) := (D^*[sk^{\mathsf{ind}}].word, m^*)$. By "strongly correct arguments" $(sk^{*\mathsf{real}}, pk^{*\mathsf{real}})$ was chosen with $\mathsf{make\_enc\_keypair}()$. "Correct key pairs" implies $\mathsf{a} \neq \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$, because otherwise $D^*[m^{\mathsf{ind}}]$ would also have got an a-handle at once. In the derived D_{a}^*, we therefore have an entry $(m^{\mathsf{hnd}}, pk^{*\mathsf{real}}, (\mathsf{honest}, sk^{*\mathsf{real}}))$ with the same distribution as id2real's choice.

"Word secrecy" for $m = pk^*$ holds since key generation does not depend on prior information, and both pk^* and sk^* become elements of Pub_Var.

7.4.10 Ciphertexts

If $type = \mathsf{enc}$, then by "strongly correct arguments", arg^{ind} is of the form $(pk^{\mathsf{ind}}, l^{\mathsf{ind}})$ with $l^{\mathsf{ind}} \neq \downarrow$. This proves the format claim in id2real. Let $(pk^*, l) := \omega^*(pk^{\mathsf{ind}}, l^{\mathsf{ind}})$, $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$, $v := \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$, and $o := D^*[m^{\mathsf{ind}}].owner$.

"Strongly correct arguments" further implies that $o \neq \mathsf{adv}$ and m^* is of the form (enc, pk, c, r) with $pk = pk^*[2]$ and r distributed as $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$.

Case 1: Ciphertext for Adversary If $v = \mathsf{a}$, no exception occurred in adv_parse and id2real obtained $arg^{\mathsf{real}} = (pk^*, l)$ as usual. Then it sets $m \leftarrow \mathsf{make\_enc}(pk^*, l)$ (for the unmodified make_enc) and $add\_arg := ()$.

For m^*, "correct arguments" implies $o = \downarrow$. Thus by "strongly correct arguments", c is distributed as $c \leftarrow \mathsf{E}_{pk}((r, l))$. Hence altogether m^* is also distributed as $\mathsf{make\_enc}(pk^*, l)$.

"Weak word secrecy" holds because only parameters m_i are used and make_enc is functional.

Case 2: Ciphertext for Honest User If $v \in \mathcal{H}$, the output of adv_parse was $arg = (pk^{\mathsf{ind}}, len)$ with $len := D^*[l^{\mathsf{ind}}].len$, and id2real derived $arg^{\mathsf{real}} = (pk^*, len)$ as claimed. Then it sets $len^* := \mathsf{list\_len}(\mathsf{nonce\_len}(k), len)$, $c' \leftarrow \mathsf{E}_{pk}(1^{len^*})$, $r' \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $m := (\mathsf{enc}, pk, c', r')$, and $add\_arg := ()$.

For m^*, "correct arguments" and $o \neq \mathsf{adv}$ imply $o = \mathsf{honest}$. Thus by "strongly correct arguments", c is distributed as $c \leftarrow \mathsf{E}_{pk}(1^{|(r,l)|})$. Here $|l| = len$ by "correct length" for $D^*[l^{\mathsf{ind}}]$, and thus $|(r, l)| = len^*$. Thus the distributions of c and c' are equal, and hence also those of m and m^*.

For "weak word secrecy", the variables used in computing m are len, constants, pk^*, and r', where len is an output of $\mathsf{TH}_{\mathcal{H}}$ and thus already in Pub_Var, pk^* is m_1 and need not be considered, and r' is a newly generated nonce r' and not stored in other variables. Hence, "weak word secrecy" holds.

7.5 Network Input from the Adversary

We now consider the effects of an input l from A at a port $\mathsf{net}_{w,u,x}?$ with $(w, u, x) \in$ *ch_from_adv*. Recall that on such an input $\mathsf{C}_{\mathcal{H}}$ acts entirely like $\mathsf{THSim}_{\mathcal{H}}$.

7.5.1 General Aspects

The length function for the port $\mathsf{net}_{w,u,x}?$, and thus the actual input, is equal in $\mathsf{Sim}_{\mathcal{H}}$, and thus $\mathsf{C}_{\mathcal{H}}$, and M'_u. Then they all increment $steps_{\mathsf{net}_{w,u,x}?}$ and only continue if l is a tagged list. Hence from now on, we assume this.

Now $\mathsf{Sim}_{\mathcal{H}}$ and thus $\mathsf{C}_{\mathcal{H}}$ call $l_{\mathsf{a}}^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ to parse the input. We first reduce our goals to properties of this algorithm and its side effects.

Lemma 7.7 To show that network inputs give equal outputs and retain all invariants, it is sufficient to prove the following properties of each call $l_{\mathsf{a}}^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ with $0 < |l| \leq \mathsf{max\_len}(k)$ and $l \in Pub\_Var$:

- At the end, $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}].word = l$ and $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}].type \not\in \{\mathsf{sks}, \mathsf{ske}\}$.
- "Correct derivation" of D_{a} and $curhnd_{\mathsf{a}}$.
- The invariants within D^* are retained, where "strongly correct arguments" is already clear and "word secrecy" need only be shown for the outermost call (without subcalls) if more entries than $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}]$ are made or updated there.

□

Proof. Correct derivation of entire state for $\mathsf{real2id}$*:* First we show that the execution of $\mathsf{real2id}$ retains the remaining derivations:

For the state of $\mathsf{TH}_{\mathcal{H}}$ this holds by definition, as well as for the buffers out_{a} and in_{a} and the corresponding counters. These buffers are empty after the macro-transition by the subroutine behavior within $\mathsf{real2id}$.

In $\mathsf{M}_{\mathcal{H}}$, nothing changes. We show that its derived counterparts D_v^* and $curhnd_v^*$ with $v \in \mathcal{H}$ are also unchanged. As $\mathsf{real2id}$ only uses local commands, handles for these users are not affected. Thus $curhnd_v^*$ is indeed unchanged, and D_v^* can change at most by changes to a non-handle attribute of an existing entry $x \in D^*$ with $x.hnd_v \neq \downarrow$. By Lemma 7.3.1, this can only happen if $x.type = \mathsf{sks}$ and in a sign command. (The changeable attribute $x.ec$ for $x.type = \mathsf{ske}$ is not involved in derivations.) However, by "key secrecy" (Lemma 4.1.6), this would imply $x.hnd_{\mathsf{a}} = \downarrow$, and we easily see below that this is not the case in such a sign command.

Nothing changes in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ because $\mathsf{M}_{\mathcal{H}}$ is not involved. The derived $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ could only change by updates to secret-key entries with honest owner, which is clearly impossible by commands entered at $\mathsf{in}_{\mathsf{a}}?$, and new entries of type enc with attribute $owner = \mathsf{honest}$, which is also impossible.

Remaining invariants for $\mathsf{real2id}$*:* "Strongly correct arguments" holds because all new entries in C^* due to inputs at $\mathsf{in}_{\mathsf{a}}?$ get an adversary handle and do not get the attribute $owner = \mathsf{honest}$, so that nothing is required. For "word secrecy", we need not consider the entry $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}]$ because the content l of $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}].word$ is already in Pub_Var by a precondition.

Actions after $\mathsf{real2id}$*:* Now we show that the remaining actions of the macro-transition retain all invariants and produce the same output in all systems. When $\mathsf{real2id}$ returns $l_{\mathsf{a}}^{\mathsf{hnd}}$, $\mathsf{Sim}_{\mathcal{H}}$ outputs and schedules $\mathsf{adv\_send\_}x(w, u, l_{\mathsf{a}}^{\mathsf{hnd}})$ at $\mathsf{in}_{\mathsf{a}}!$. Let $l^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}].ind$. Then $\mathsf{TH}_{\mathcal{H}}$ verifies that $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$; this is true by "correct arguments" because $D^*[l^{\mathsf{ind}}].word = l$ is a tagged list.

If $l_u^{\mathsf{hnd}} := D^*[l^{\mathsf{ind}}].hnd_u \neq \downarrow$, $\mathsf{TH}_{\mathcal{H}}$ outputs $(w, x, l_u^{\mathsf{hnd}})$ at $\mathsf{out}_u!$. By "correct derivation" of D_u, M'_u finds the same handle when executing $(l_u^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$ and makes the same output. Then all invariants are clearly maintained.

Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds this handle as $l_u^{\mathsf{hnd}} := curhnd_u\text{++}$. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word l, and thus makes a new entry $(l_u^{\mathsf{hnd}}, l, \mathsf{list}, ())$ with the same handle. (The test $|l| \leq \mathsf{max\_len}(k)$ in $:\leftarrow$ is true by the initial domain test.) "Correct derivation" is clear. By Lemma 7.3.3, this is all we have to show. ■

7.5.2 General Aspects of real2id

We show inductively that for every $m \in \{0,1\}^+$ with $|m| \leq \mathsf{max\_len}(k)$ and $m \in Pub\_Var$, the execution of $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$ together with its side effects fulfills the conditions from Lemma 7.7.

If there is already a handle m^{hnd} with $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, real2id returns that. The output condition of Lemma 7.7 is fulfilled by "correct derivation", and the others because no state changes are made.

Otherwise, the word m is not yet present in D_{a}. Then id2real sets $(type, arg) := \mathsf{parse}(m)$. This yields $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}\}$. As parse is a functional algorithm, no invariants are affected. Then id2real calls an algorithm $add\_arg \leftarrow \mathsf{real2id\_}type(m, arg)$ with side-effects. Finally it sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$.

We therefore have to show the properties from Lemma 7.7 for these type-specific algorithms together with those last two assignments. In the following, we write arg in the form it must have in an output $(type, arg)$ of parse.

7.5.3 Garbage

The algorithm $\mathsf{real2id\_garbage}(m, ())$ calls $m^{\mathsf{hnd}} \leftarrow \mathsf{adv\_garbage}(|m|)$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$. Then $\mathsf{TH}_{\mathcal{H}}$ makes a new entry with $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$. Together with the new entry in D_{a}, this results in $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{garbage}, arg := (), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := |m|, word := m)$. (Attributes $parsed_u$ are only defined for $u \in \mathcal{H}$ and are all still $\downarrow$.)

"Correct derivation" is clearly retained, and the new entry is as required in Lemma 7.7. "Fully defined", "correct length", "no unparsed secret keys", and "correct arguments" are obvious. "Word secrecy" is clear as no other entries are involved.

Now assume that "word uniqueness" were not fulfilled. Then there is a prior entry $x \in D^*$ with $x.word = m$ and $x.type \notin \{\mathsf{sks}, \mathsf{ske}\}$. Further, it has $x.hnd_{\mathsf{a}} = \downarrow$ because the word m does not exist in D_{a}. By "correct arguments", it has $x.type \in \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$. This leaves only the case garbage, but that is impossible because such entries are only made due to adversary commands and thus with $x.hnd_{\mathsf{a}} \neq \downarrow$.

7.5.4 Data

When the algorithm $\mathsf{real2id\_data}(m, (m'))$ is called, we know that $m = (\mathsf{data}, m')$.

It simply calls $m^{\mathsf{hnd}} \leftarrow \mathsf{store}(m')$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$. $\mathsf{TH}_{\mathcal{H}}$ accepts this input because $|m'| < |m| \leq \mathsf{max\_len}(k)$ by the preconditions. Then $\mathsf{TH}_{\mathcal{H}}$ checks whether there is an m^{ind} with $D^*[m^{\mathsf{ind}}].type = \mathsf{data}$ and $D^*[m^{\mathsf{ind}}].arg = (m')$. By "correct arguments", this is equivalent to $D^*[m^{\mathsf{ind}}].word = m$. If this entry exists, it has no a-handle yet, because that would contradict the absence of m in D_{a} by "correct derivation". Thus $\mathsf{TH}_{\mathcal{H}}$ assigns $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$. "Correct derivation" together with the new entry in D_{a} is then clear, and the other conditions of Lemma 7.7 clearly hold.

Otherwise, $\mathsf{TH}_\mathcal{H}$ verifies $\mathsf{data\_len}^*(|m'|) \leq \mathsf{max\_len}(k)$. This holds because $\mathsf{data\_len}^*(|m'|) = |m|$. Then $\mathsf{TH}_\mathcal{H}$ makes a new entry with $m^{\mathsf{hnd}} := \mathit{curhnd}_\mathsf{a}\text{++}$. Together with the new entry in D_a, this results in $D^* :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{data}, \mathit{arg} := (m'), \mathit{hnd}_\mathsf{a} := m^{\mathsf{hnd}}, \mathit{len} := \mathsf{data\_len}^*(|m'|), \mathit{word} := m)$. "Correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. "Fully defined", "correct length", and "no unparsed secret keys" are obvious. "Word uniqueness" holds by the preconditions of this case. "Correct arguments" was already written out above. "Word secrecy" is clear as no other entries are involved.

7.5.5 Lists

When the algorithm $\mathsf{real2id\_list}(m, (m_1, \ldots, m_j)$ is called, we know that $m = (\mathsf{list}, m_1, \ldots, m_j)$ with $m_i \in \{0,1\}^+$ for all i. The precondition $|m| \leq \mathsf{max\_len}(k)$ implies $j \leq \mathsf{max\_len}(k)$ and $|m_i| \leq \mathsf{max\_len}(k)$ for all i. Further $m_i \in \mathit{Pub\_Var}$ for all i because they have been generated from $m \in \mathit{Pub\_Var}$ by the functional algorithm parse.

Hence when $\mathsf{real2id\_list}$ starts with recursive calls $m_i^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m_i)$ for $i := 1, \ldots, j$, these calls fulfill the preconditions of Lemma 7.7. Thus each call retains all the invariants and ensures $D^*[\mathit{hnd}_\mathsf{a} = m_i^{\mathsf{hnd}}].\mathit{word} = m_i$ for $i = 1, \ldots, j$. Let $m_i{}^{\mathsf{ind}} := D^*[\mathit{hnd}_\mathsf{a} = m_i^{\mathsf{hnd}}].\mathit{ind}$.

Then $\mathsf{real2id\_list}$ calls $m^{\mathsf{hnd}} \leftarrow \mathsf{list}(m_1^{\mathsf{hnd}}, \ldots, m_j^{\mathsf{hnd}})$ at $\mathsf{in}_\mathsf{a}!$ and sets $\mathit{add\_arg} := ()$. $\mathsf{TH}_\mathcal{H}$'s domain expectations $j \leq \mathsf{max\_len}(k)$ and $m_i^{\mathsf{ind}} \neq \downarrow$ are fulfilled, and $D^*[m_i^{\mathsf{ind}}].\mathit{type} \notin \{\mathsf{sks}, \mathsf{ske}\}$ is also a consequence of Lemma 7.7.

Thus $\mathsf{TH}_\mathcal{H}$ checks whether there is an m^{ind} with $D^*[m^{\mathsf{ind}}].\mathit{type} = \mathsf{list}$ and $D^*[m^{\mathsf{ind}}].\mathit{arg} = (m_1^{\mathsf{ind}}, \ldots, m_j^{\mathsf{ind}})$. By "correct arguments", this is equivalent to $D^*[m^{\mathsf{ind}}].\mathit{word} = m$. If this entry exists, it has no a-handle yet, because that would contradict the absence of m in D_a by "correct derivation". Thus $\mathsf{TH}_\mathcal{H}$ assigns one as $m^{\mathsf{hnd}} := \mathit{curhnd}_\mathsf{a}\text{++}$. "Correct derivation" together with the new entry in D_a is then clear, and the other conditions of Lemma 7.7 clearly hold.

Otherwise, $\mathsf{TH}_\mathcal{H}$ sets $\mathit{length} := \mathsf{list\_len}^*(D^*[m_1^{\mathsf{ind}}].\mathit{len}, \ldots, D^*[m_j^{\mathsf{ind}}].\mathit{len})$ and verifies $\mathit{length} > \mathsf{max\_len}(k)$. This holds because by "correct length" $\mathit{length} = \mathsf{list\_len}^*(|m_1|, \ldots, |m_j|) = |m|$. Then $\mathsf{TH}_\mathcal{H}$ makes a new entry with $m^{\mathsf{hnd}} := \mathit{curhnd}_\mathsf{a}\text{++}$. Together with the new entry in D_a, this results in $D^* :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{list}, \mathit{arg} := (m_1^{\mathsf{ind}}, \ldots, m_j^{\mathsf{ind}}), \mathit{hnd}_\mathsf{a} := m^{\mathsf{hnd}}, \mathit{len} := \mathit{length}, \mathit{word} := m)$.

"Correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. "Fully defined" and "no unparsed secret keys" are obvious. "Word uniqueness" holds by the preconditions of this case. "Correct length" was already shown, and "correct arguments" was written out above. "Word secrecy" is clear as no other entries are involved.

7.5.6 Nonces

The algorithm $\mathsf{real2id\_nonce}(m, ())$ calls $m^{\mathsf{hnd}} \leftarrow \mathsf{gen\_nonce}()$ at $\mathsf{in}_\mathsf{a}!$ and sets $\mathit{add\_arg} := ()$. Then $\mathsf{TH}_\mathcal{H}$ makes a new entry with $m^{\mathsf{hnd}} := \mathit{curhnd}_\mathsf{a}\text{++}$. Together with the new entry in D_a, this results in $D^* :\Leftarrow (\mathit{ind} := \mathit{size}\text{++}, \mathit{type} := \mathsf{nonce}, \mathit{arg} := (), \mathit{hnd}_\mathsf{a} := m^{\mathsf{hnd}}, \mathit{len} := \mathsf{nonce\_len}^*(k), \mathit{word} := m)$.

"Correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. "Fully defined", "no unparsed secret keys", and "correct arguments" are obvious. For "correct length", the tests in parse imply that m is of this length. "Word secrecy" is clear as no other entries are involved.

For "word uniqueness", assume there is a prior entry $x \in D^*$ with $x.\mathit{word} = m$. We then put this run in the error set $\mathit{Nonce\_Guess}$. We have $x.\mathit{hnd}_\mathsf{a} = \downarrow$ by "correct derivation" of D_a, because m is not present in D_a. Thus $x.\mathit{word} \notin \mathit{Pub\_Var}$. By "correct arguments", clearly also $x.\mathit{type} = \mathsf{nonce}$.

7.5.7 Public Signature Keys

The algorithm $\mathsf{real2id\_pks}(m, ())$ calls $(sk^{\mathsf{hnd}}, m^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_sig\_keypair}()$ at $\mathsf{in_a}!$ and sets $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}\text{++}, \epsilon, ())$ for the secret key and $add\_arg := (\mathsf{adv})$ for the public key.

$\mathsf{TH}_{\mathcal{H}}$ then also makes two new entries with $sk^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$ and $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$. In $\mathsf{C}_{\mathcal{H}}$, the secret-key entry results in $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{sks}, arg := (0), hnd_{\mathsf{a}} := sk^{\mathsf{hnd}}, len := 0, word := \epsilon)$. Here "correct derivation" and "fully defined" are clear, nothing is required under "word uniqueness", "correct length", and "correct arguments", nor under "no unparsed secret keys" because the owner is a. "Word secrecy" for this additional entry holds because $D^*[hnd_{\mathsf{a}} = sk^{\mathsf{hnd}}].word = \epsilon$ is a constant.

The public-key entry results in $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{pks}, arg := (), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := \mathsf{pks\_len}^*(k), word := m)$. It fulfills the output conditions of Lemma 7.7. Let $pk^{\mathsf{ind}} := size$. Then "correct derivation" holds because $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$ designates the secret-key entry with $\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a}$, so that $add\_arg = (\mathsf{adv})$ is the correct choice in D_{a}. "Fully defined", "no unparsed secret keys", and "correct arguments" are obvious. For "correct length", the tests in parse imply that m is of this length. "Word secrecy" need not be shown for this entry.

For "word uniqueness", assume there is a prior entry $x \in D^*$ with $x.word = m$. We then put this run in the error set Key_Guess. We have $x.hnd_{\mathsf{a}} = \downarrow$ by "correct derivation" of D_{a}, because m is not present in D_{a}. Thus $x.word \notin Pub\_Var$.

7.5.8 Signatures

When $\mathsf{real2id\_sig}(m, (pk^*, l))$ is called, we know from parsing that pk^* is of the form (pks, pk) and $|pk^*| = \mathsf{pks\_len}^*(k) \leq \mathsf{max\_len}(k)$. Further, l is a tagged list and shorter than m, so that also $|l| \leq \mathsf{max\_len}(k)$. Moreover, $pk^*, l \in Pub\_Var$ because they were generated from $m \in Pub\_Var$ by the functional algorithm parse.

Hence when $\mathsf{real2id\_sig}$ starts with recursive calls $pk^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(pk^*)$ and $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$, these calls fulfill the preconditions of Lemma 7.7. Thus they retain all invariants and ensure $D^*[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}}].word = pk^*$ and $D^*[hnd_{\mathsf{a}} = l^{\mathsf{hnd}}].word = l$.

Let $pk^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}}].ind$, $l^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = l^{\mathsf{hnd}}].ind$, and $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$. The consequences on pk^* and l from parsing m imply that they parse to type pks and list, respectively. Hence "correct arguments" implies $D^*[pk^{\mathsf{ind}}].type = \mathsf{pks}$ and $D^*[l^{\mathsf{ind}}].type = \mathsf{list}$, and "correct key pairs" implies $D^*[sk^{\mathsf{ind}}].type = \mathsf{sks}$.

Case 1: Adversary Key If $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{adv})$, then $\mathsf{real2id\_sig}$ sets $sk^{\mathsf{hnd}} := pk^{\mathsf{hnd}} - 1$, calls $m^{\mathsf{hnd}} \leftarrow \mathsf{sign}(sk^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

By "correct derivation" for pk^{hnd}, this case implies $\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a}$. With "correct key pairs" this further implies $D^*[sk^{\mathsf{ind}}].hnd_{\mathsf{a}} = D^*[pk^{\mathsf{ind}}].hnd_{\mathsf{a}} - 1 = sk^{\mathsf{hnd}}$.

Thus the input passes $\mathsf{TH}_{\mathcal{H}}$'s type checks. Next $\mathsf{TH}_{\mathcal{H}}$ sets $length := \mathsf{sig\_len}^*(k, D^*[l^{\mathsf{ind}}].len)$ and verifies $length \leq \mathsf{max\_len}(k)$. This holds because $|m| \leq \mathsf{max\_len}(k)$ and we know from parsing that $|m| = \mathsf{list\_len}(|\mathsf{sig}|, \mathsf{pks\_len}'(k), \mathsf{nonce\_len}(k), |l|, \mathsf{sig\_len}'(k, |l|)) = \mathsf{sig\_len}^*(k, |l|)$ and from "correct length" that $D^*[l^{\mathsf{ind}}].len = |l|$. As $\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a}$, $\mathsf{TH}_{\mathcal{H}}$ does not verify the counter size.

Hence $\mathsf{TH}_{\mathcal{H}}$ sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\text{++}$, obtains pk^{ind} as $sk^{\mathsf{ind}} + 1$, updates the signature counter and makes a new entry. Together with the new entry in D_{a}, this results in $c := D^*[sk^{\mathsf{ind}}].arg[1]\text{++}$ and $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{sig}, arg := (pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := length, word := m)$.

The change in the entry $D^*[sk^{\mathsf{ind}}]$ retains "correct derivation". Among the invariants, it can only affect "correct arguments", but for secret keys nothing is required, and "word secrecy", but the update information is from within $\mathsf{TH}_{\mathcal{H}}$ and A.

We now consider the signature entry. "Correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. "Fully defined" and "no unparsed secret keys" are obvious, and so is "correct arguments" given the special definition for type sig. "Correct length" was already shown above. "Word secrecy" need not be shown for this entry.

Finally, we prove "word uniqueness": Assume there were a prior entry $x \in D^*$ with $x.word = m$. It has $x.hnd_{\mathsf{a}} = \downarrow$ because the word m does not exist in D_{a}. However, such an adversary signature without a-handle is impossible: "Correct arguments" and "word uniqueness" for pk^* and l imply $x.type = \mathsf{sig}$ and $x.arg = (pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c')$ for some c'. Such entries arise only from commands sign and $\mathsf{adv\_transform\_sig}$ in $\mathsf{TH}_{\mathcal{H}}$ and thus in $\mathsf{C}_{\mathcal{H}}$. The latter always gives an a-handle, and so does the former if input at $\mathsf{in_a}?$. If input at $\mathsf{in}_u?$ with $u \in \mathcal{H}$, it needs an argument sk'^{hnd}_u where $sk'^{\mathsf{ind}} + 1 = pk^{\mathsf{ind}}$ for $sk'^{\mathsf{ind}} := D^*[hnd_u = sk'^{\mathsf{hnd}}_u].ind$. This implies $sk'^{\mathsf{ind}} = sk^{\mathsf{ind}}$, and this contradicts "key secrecy" because we know the handle $D^*[sk^{\mathsf{ind}}].hnd_{\mathsf{a}} = sk^{\mathsf{hnd}}$.

Case 2: Transformed Signature Now let $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk^*)$. By "correct derivation" for pk^{hnd}, this implies $v := \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$. Further, m can be written as $(\mathsf{sig}, pk, r, l, sig)$ by parsing.

Now assume there exist s^{hnd}, r', and sig' with $m' := D_{\mathsf{a}}[s^{\mathsf{hnd}}].word = (\mathsf{sig}, pk, r', l, sig')$ and $D_{\mathsf{a}}[s^{\mathsf{hnd}}].type = \mathsf{sig}$. Then $\mathsf{real2id\_sig}$ calls $m^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_sig}(s^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

Let $s^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = s^{\mathsf{hnd}}].ind$. By "correct derivation", we have $D^*[s^{\mathsf{ind}}].type = \mathsf{sig}$. With the preconditions about m', "correct arguments" for s^{ind}, and "word uniqueness" for pk^* and l, this implies $D^*[s^{\mathsf{ind}}].arg = (pk^{\mathsf{ind}}, l^{\mathsf{ind}}, c)$ for some c.

Hence $\mathsf{TH}_{\mathcal{H}}$ sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$ and makes a new entry. Together with the new entry in D_{a}, this results in $D^* :\Leftarrow (ind := size{++}, type := \mathsf{sig}, arg := (pk^{\mathsf{ind}}, l^{\mathsf{ind}}, \mathsf{false}), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := D^*[s^{\mathsf{ind}}].len, word := m)$.

"Correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. "Fully defined" and "no unparsed secret keys" are obvious. "Correct arguments" holds because we showed that the arguments copied from $D^*[s^{\mathsf{ind}}]$ are those that we get by parsing m. "Word uniqueness" is shown exactly as in Case 1. For "correct length", we use that $D^*[s^{\mathsf{ind}}].len = |m'|$ by "correct length" for s^{ind}. Thus we only have to show $|m| = |m'|$. This holds because both parse as signatures with the same component l. "Word secrecy" need not be shown for this entry.

Case 3: Forged Signature In the final case, we have $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk^*)$ and thus $v := \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$ as in Case 2 and again $m =: (\mathsf{sig}, pk, r, l, sig)$. However, now there exist no s^{hnd}, r', and sig' with $D_{\mathsf{a}}[s^{\mathsf{hnd}}].word = (\mathsf{sig}, pk, r', l, sig')$ and $D_{\mathsf{a}}[s^{\mathsf{hnd}}].type = \mathsf{sig}$. Here $\mathsf{Sim}_{\mathcal{H}}$ gives up. We distinguish two cases.

If there exists an entry $x \in D^*$ and sig' with $x.word = (\mathsf{sig}, pk, r, l, sig')$ and $x.type = \mathsf{sig}$, we put the run in the error set $Nonce\_Guess$. (The adversary has guessed the nonce-part r.) We have $x.hnd_{\mathsf{a}} = \downarrow$ by "correct derivation" of D_{a}, because in Case 3 no such word is present in D_{a}. Thus $x.word \notin Pub\_Var$.

Otherwise, we put the run in the error set $Forge$ and designate the forgery $(pk, (r, l), sig)$. Note that $\mathsf{test}_{pk}(sig, (r, l)) = \mathsf{true}$ because this was verified when parsing m, and that $v = \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$. Further, "key secrecy" and "strongly correct arguments" for $D^*[sk^{\mathsf{ind}}]$ imply that pk was chosen in $\mathsf{gen\_sig\_keypair}$ together with $sk := D^*[sk^{\mathsf{ind}}].word[2]$, and thus as $(sk, pk) \leftarrow \mathsf{gen_S}(1^k, 1^{\mathsf{max\_skc}(k)})$.

7.5.9 Public Encryption Keys

The algorithm $\mathsf{real2id\_pke}(m, ())$ calls $(sk^{\mathsf{hnd}}, m^{\mathsf{hnd}}) \leftarrow \mathsf{gen\_enc\_keypair}()$ at $\mathsf{in_a}!$ and sets $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}{++}, \epsilon, ())$ for the secret key and $add\_arg := (\mathsf{adv})$ for the public key.

$\mathsf{TH}_{\mathcal{H}}$ then also makes two new entries with $sk^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$ and $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$. In $\mathsf{C}_{\mathcal{H}}$, the secret-key entry results in $D^* :\Leftarrow (ind := size{++}, type := \mathsf{ske}, arg := (), hnd_{\mathsf{a}} := sk^{\mathsf{hnd}}, len := 0, word := \epsilon)$. Here "correct derivation" and "fully defined" are clear. Nothing is required under "word uniqueness", "correct length", and "correct arguments", nor under "no unparsed secret keys" because the owner is a. "Word secrecy" for this additional entry holds because $D^*[hnd_{\mathsf{a}} = sk^{\mathsf{hnd}}].word = \epsilon$ is a constant.

The public-key entry results in $D^* :\Leftarrow (ind := size{++}, type := \mathsf{pke}, arg := (), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := \mathsf{pke\_len}^*(k), word := m)$. It fulfills the output conditions of Lemma 7.7. Let $pk^{\mathsf{ind}} := size$. Then "correct derivation" holds because $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$ designates the secret-key entry with $\mathsf{owner}(D^*[sk^{\mathsf{ind}}]) = \mathsf{a}$, so that $add\_arg = (\mathsf{adv})$ is the correct choice in D_{a}. "Fully defined", "no unparsed secret keys", and "correct arguments" are obvious. For "correct length", the tests in parse imply that m is of this length. "Word secrecy" need not be shown for this entry.

For "word uniqueness", assume there is a prior entry $x \in D^*$ with $x.word = m$. We then put this run in the error set *Key_Guess*. We have $x.hnd_{\mathsf{a}} = \downarrow$ by "correct derivation" of D_{a}, because m is not present in D_{a}. Thus $x.word \notin Pub\_Var$.

7.5.10 Ciphertexts

When $\mathsf{real2id\_enc}(m, (pk^*))$ is called, we know from parsing that $pk^* = (\mathsf{pke}, pk)$ with $|pk^*| = \mathsf{pke\_len}^*(k) \leq \mathsf{max\_len}(k)$. Further, $pk^* \in Pub\_Var$ because it was generated from $m \in Pub\_Var$ by the functional algorithm parse.

Hence when $\mathsf{real2id\_enc}$ starts with a recursive call $pk^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(pk^*)$, this call fulfills the preconditions of Lemma 7.7. Thus it retains all invariants and ensures $D^*[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}}].word = pk^*$.

Let $pk^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = pk^{\mathsf{hnd}}].ind$ and $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} - 1$. The consequences on pk^* from parsing m imply that it parses to type pke. Hence "correct arguments" implies $D^*[pk^{\mathsf{ind}}].type = \mathsf{pke}$, and "correct key pairs" implies $D^*[sk^{\mathsf{ind}}].type = \mathsf{ske}$. Let $v := \mathsf{owner}(D^*[sk^{\mathsf{ind}}])$.

In all cases of $\mathsf{real2id\_enc}$, only the entry for m^{hnd} is made, and always with word m and type enc; further always $add\_arg = ()$. Thus "correct derivation" is clearly retained, and the output condition of Lemma 7.7 fulfilled. Further, "fully defined" and "no unparsed secret keys" are obvious, and "word secrecy" need not be shown for this entry. For "word uniqueness", we put the run in the error set *Nonce_Guess* if there is a prior entry $x \in D^*$ with $x.word = m$. (The adversary has guessed the nonce-part r of m.) We have $x.hnd_{\mathsf{a}} = \downarrow$ because the word m does not exist in D_{a}. Thus $x.word \notin Pub\_Var$. By "correct arguments", clearly also $x.type = \mathsf{enc}$.

Hence in the individual cases, "correct length" and "correct arguments" remain to be shown.

Case 1: Adversary Key If $v = \mathsf{adv}$, then "correct derivation" implies $D_{\mathsf{a}}[pk^{\mathsf{hnd}}].add\_arg = (\mathsf{adv})$. Hence $\mathsf{real2id\_enc}$ calls $m^{\mathsf{hnd}} \leftarrow \mathsf{adv\_invalid\_ciph}(pk^{\mathsf{hnd}}, |m|)$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

This input passes $\mathsf{TH}_{\mathcal{H}}$'s domain and type check. Hence $\mathsf{TH}_{\mathcal{H}}$ sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$ and makes a new entry. Together with the new entry in D_{a}, this results in $D^* :\Leftarrow (ind := size{++}, type := \mathsf{enc}, arg := (pk^{\mathsf{ind}}), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := |m|, word := m)$.

"Correct length" is obvious. For "correct arguments", the general requirement for the public-key argument is clearly fulfilled. The fact $o := D^*[size].owner = \downarrow$ is correct for $v = \mathsf{a}$, and nothing else needs to be shown in this case.

Case 2: Invalid Ciphertext for Honest User If $v \in \mathcal{H}$, then "correct derivation" implies $D_\mathsf{a}[pk^\mathsf{hnd}].add\_arg = (\mathsf{honest}, sk^*)$ with $sk^* = \omega(sk^\mathsf{ind})$. Thus $\mathsf{real2id\_enc}$ computes $l := \mathsf{parse\_decrypt}(sk^*, m)$.

For this case, assume $l = \downarrow$. Then $\mathsf{real2id\_enc}$ calls $c^\mathsf{hnd} \leftarrow \mathsf{adv\_invalid\_ciph}(pk^\mathsf{hnd}, |m|)$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

As above, this input passes $\mathsf{TH}_\mathcal{H}$'s domain and type check, and $\mathsf{TH}_\mathcal{H}$ sets $m^\mathsf{hnd} := curhnd_\mathsf{a}$++ and makes a new entry. Together with the new entry in D_a, this results in $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{enc}, arg := (pk^\mathsf{ind}), hnd_\mathsf{a} := m^\mathsf{hnd}, len := |m|, word := m, owner := \mathsf{adv})$.

"Correct length" is obvious. For "correct arguments", the general requirement for the public-key argument is clearly fulfilled. The fact $o := D^*[size].owner = \mathsf{adv}$ is correct for $v \in \mathcal{H}$. The resulting requirement on decryption is fulfilled by construction because $l^\mathsf{ind} := arg^\mathsf{ind}[2] = \downarrow$, and $l := \mathsf{parse\_decrypt}(sk^*, m) = \downarrow$ is the precondition of this case.

Case 3: Valid Ciphertext for Honest User Let $v \in \mathcal{H}$ again and sk^* and l be defined as in Case 2, but now $l \neq \downarrow$.

Then $\mathsf{real2id\_enc}$ makes a recursive call $l^\mathsf{hnd} \leftarrow \mathsf{real2id}(l)$. We have to show that this call fulfills the preconditions of Lemma 7.7. First, $|l| \leq \mathsf{max\_len}(k)$ because the construction of $\mathsf{parse\_decrypt}$ implies $|l| \leq |m|$. Secondly, l only depends on sk^* and m. By a precondition, $m \in Pub\_Var$. Further, by the definition of Pub_Var, the existence of $D^*[pk^\mathsf{ind}].hnd_\mathsf{a}$ implies that also $sk^* \in Pub\_Var$. Thus also $l \in Pub\_Var$.

Thus the recursive call retains all invariants and ensures $D^*[hnd_\mathsf{a} = l^\mathsf{hnd}].word = l$. Now $\mathsf{real2id\_enc}$ calls $c^\mathsf{hnd} \leftarrow \mathsf{encrypt}(pk^\mathsf{hnd}, l^\mathsf{hnd})$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

The arguments pass $\mathsf{TH}_\mathcal{H}$'s domain check and the type check for pk^hnd. Further, l is a tagged list by the tests in $\mathsf{parse\_decrypt}$, and thus by "correct arguments", $D^*[l^\mathsf{ind}].type = \mathsf{list}$ for $l^\mathsf{ind} := D^*[hnd_\mathsf{a} = l^\mathsf{hnd}].ind$. Thus $\mathsf{TH}_\mathcal{H}$ sets $length := \mathsf{enc\_len}^*(k, D^*[l^\mathsf{ind}].len)$, where $D^*[l^\mathsf{ind}].len = |l|$. The tests made when parsing m and in $\mathsf{parse\_decrypt}$ imply that $length = |c| \leq \mathsf{max\_len}(k)$. Thus $\mathsf{TH}_\mathcal{H}$ sets $m^\mathsf{hnd} := curhnd_\mathsf{a}$++ and makes a new entry. Together with the new entry in D_a, this results in $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{enc}, arg := (pk^\mathsf{ind}, l^\mathsf{ind}), hnd_\mathsf{a} := m^\mathsf{hnd}, len := |m|, word := length, owner := \mathsf{adv})$.

"Correct length" was just shown. For "correct arguments", the general requirement for the public-key argument is clearly fulfilled. The fact $o := D^*[size].owner = \mathsf{adv}$ is correct for $v \in \mathcal{H}$. The resulting requirement on decryption is fulfilled by construction because indeed $\omega(l^\mathsf{ind}) = l$ where l was defined as $l := \mathsf{parse\_decrypt}(sk^*, m)$.

7.6 Scheduling of a Secure Channel

Finally, we consider a scheduling signal i from A for a secure channel, i.e., for $(w, u, x) \in ch\_honest$. By "correct derivation", we have $net_{w,u,x} = \omega^*(net\_id_{w,u,x})$, where the former is the buffer in $\mathsf{M}_\mathcal{H}$ and the latter in $\mathsf{THSim}_\mathcal{H}$ and $\mathsf{C}_\mathcal{H}$. This implies that the buffers are of the same length i'. If $i > i'$, nothing happens. Otherwise, $\mathsf{TH}_\mathcal{H}$ gets the input $l^\mathsf{ind} := net\_id_{w,u,x}[i]$ and M'_u gets $l := net_{w,u,x}[i] = D^*[l^\mathsf{ind}.word]$. By "message correctness" (recall Lemma 4.1), we have $D[l^\mathsf{ind}].type = \mathsf{list}$, and thus by "correct arguments", l is a tagged list. Further, $|l| \leq \mathsf{max\_len}(k)$ by "correct length" and "length bounds". Hence M'_u accepts l.

If $D^*[l^{\mathsf{ind}}].hnd_u$ already exists, both return it. Otherwise $\mathsf{TH}_{\mathcal{H}}$ adds it as $l^{\mathsf{hnd}} := curhnd_u{++}$. By "word uniqueness" and "correct derivation", M'_u does not find another entry with the word l, and thus makes a new entry $(l^{\mathsf{hnd}}, l, \mathsf{list}, ())$ with the same handle. "Correct derivation" is clear. By Lemma 7.3.3, this is all we have to show.

7.7 Error Sets

We now show that the union of all error sets has negligible probability. More precisely, this means sequences of error sets, indexed by the basic security parameter k, such as $(Forge_k)_{k\in\mathbb{N}}$. We continue to omit the parameter k when it is irrelevant. The proofs rely on the security of the cryptographic primitives.

Recall that we had error sets *Nonce_Coll*, *Key_Coll*, *Ciph_Coll*, *Nonce_Guess*, *Key_Guess*, and *Forge*. This gives a constant number of sequences. Hence, if each sequence has negligible probability, then so has the sequence of the set unions. Hence we now assume for contradiction that one sequence has a larger probability for certain polynomial-time users H and adversary A.

Recall that the elements of the error sets are runs of the combined system $\mathsf{C}_{\mathcal{H}}$. The proofs rely on the fact that the execution of $\mathsf{C}_{\mathcal{H}}$ with H and A is polynomial-time. $\mathsf{C}_{\mathcal{H}}$ is even polynomial-time on its own, because $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$ are polynomial-time by Lemmas 4.2.1, 6.1, and 7.2.1, and essentially all actions of $\mathsf{C}_{\mathcal{H}}$ occur in one of these systems.

7.7.1 Nonce Collisions

The error set *Nonce_Coll* occurs in Sections 7.3.6, 7.3.7, and 7.3.8 for the generation of "official" nonces, and the nonce-components r in ciphertexts and signatures. A run is put into this set if a new nonce, created randomly as $r \overset{\mathcal{R}}{\leftarrow} \{0,1\}^{\mathsf{nonce\_len}(k)}$, matches an already existing value. Hence for every pair of a new nonce and an old value, the success probability is bounded by $2^{-\mathsf{nonce\_len}(k)}$, which we required to be negligible. As there are only polynomially many such pairs, the overall probability is also negligible.

7.7.2 Key Collisions

The error set *Key_Coll* occurs in Sections 7.3.7 and 7.3.8 for the generation of public signature and encryption keys. We first consider encryption keys. A run is put into this set if a new public key pk, created correctly with $(sk, pk) \leftarrow \mathsf{gen}_{\mathsf{E}}(1^k)$, matches an already existing value, which may have been chosen by the adversary.

Let $P_{max,k}$ be the maximum probability of any particular value of pk for security parameter k. Then $P_{max,k}$ is an upper bound for the collision probability of every pair of an old value and new public key. As there are only polynomially many such pairs, it is sufficient to prove that $P_{max,k}$ is negligible. Assume for contradiction it were not.

Let an adversary $\mathsf{A}_{\mathsf{enc}}$ on the encryption system, given a key pk, independently generate a key pair $(sk', pk') \leftarrow \mathsf{gen}_{\mathsf{E}}(1^k)$. Then the probability of $pk' = pk$ is at least ${P_{max,k}}^2$, which is still not negligible. In this event, we have $m = \mathsf{D}_{sk'}(c)$ for every ciphertext $c \leftarrow \mathsf{E}_{pk}(m)$, i.e., the adversary can decrypt ciphertexts intended for the honest participant, which clearly contradicts the security of the encryption system.

The argument for signature keys is completely analogous, where signatures made with sk' pass the test with pk.

7.7.3 Ciphertext Collisions

The error set *Ciph_Coll* occurs in Section 7.3.8 for the basic command $\mathsf{decrypt}$. A run is put into this set if the encryption machine $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ has generated two entries (l_1^*, pk, c) and (l_2^*, pk, c) with the same public and ciphertext, but $l_1^* \neq l_2^*$, in *ciphers*. In the real encryption machine $\mathsf{Enc}_{\mathcal{H}}$, this situation is impossible because the ciphertexts are actually under the messages l_1^* and l_2^* (while in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ they are under fixed messages), and decryption is unique for correctly generated key pairs. Indistinguishability of $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$ and $\mathsf{Enc}_{\mathcal{H}}$, and the fact that the content of *ciphers* is visible externally, implies that the probability of the same event is at most negligible in $\mathsf{Enc}_{\mathsf{sim},\mathcal{H}}$.

7.7.4 Nonce Guessing

The error set *Nonce_Guess* occurs in Section 7.5.6, 7.5.8, and 7.5.10. A run is put into this set if the adversary has guessed an existing nonce value that ideally he should not have seen. In all these cases we showed that the adversary had guessed the word of an entry $x \in D^*$ with $x.hnd_{\mathsf{a}} = \downarrow$, $x.word \not\in Pub\_Var$, and $x.type \in \{\mathsf{nonce}, \mathsf{sig}, \mathsf{enc}\}$.

"Strongly correct arguments" applies to these entries. Thus each of them contains a nonce part generated as $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$ independently of everything else.

"Word secrecy" means that no information flowed from r into Pub_Var, which is a superset of the information known to the adversary A. Hence for one guess at one value, the success probability is $2^{-\mathsf{nonce\_len}(k)}$ and thus negligible, and there are only a polynomially many values and polynomially many opportunities of guessing.

7.7.5 Key Guessing

The error set *Key_Guess* occurs in Sections 7.5.7 and 7.5.9. A run is put into this set if the adversary has guessed an existing public key that ideally he should not have seen. Similar to nonce guessing, we showed that the adversary had guessed the word of an entry $x \in D^*$ with $x.hnd_{\mathsf{a}} = \downarrow$, $x.word \not\in Pub\_Var$, and $x.type \in \{\mathsf{pks}, \mathsf{pke}\}$.

As above, word secrecy implies that the adversary had no information at all about the component $pk := x.word[2]$. Hence for one guess at one value, the success probability is bounded by $P_{max,k}$ as in Section 7.7.2 and thus negligible, and there are only a polynomially many guesses and values.

7.7.6 Signature Forgery

The error set *Forge* occurs in Section 7.5.8 for signature forgeries. In the runs put into this set we designated a triple $(pk, (r,l), sig)$ as a forgery and showed that $\mathsf{test}_{pk}(sig, (r,l)) = \mathsf{true}$ for a key pair chosen correctly as $(sk, pk) \leftarrow \mathsf{gen}_{\mathsf{S}}(1^k, 1^{\mathsf{max\_skc}(k)})$.

In the combined system $\mathsf{C}_{\mathcal{H}}$, this secret key sk was a component $D^*[sk^{\mathsf{ind}}].word[2]$ with $v = \mathsf{owner}(D^*[sk^{\mathsf{ind}}]) \in \mathcal{H}$. Thus it is only used if the command sign is entered at port $\mathsf{in}_v?$, and there within normal signing operations $sig \leftarrow \mathsf{sign}_{sk,c}((r,l))$, executed with an internal signature counter c which is incremented with each signature and not used otherwise.

Further, if (r,l) had ever been signed with sk before, the command sign would lead to an entry $x \in D^*$ with $x.type = \mathsf{sig}$ and $x.word$ of the form $(\mathsf{sig}, pk, r, l, sig')$. However, the existence of such an entry was excluded in the conditions for putting the run in the set *Forge*. Thus we have indeed a valid forgery for the underlying signature system.

This argument was almost a rigorous reduction proof already: We construct an adversary $\mathsf{A}_{\mathsf{sig}}$ against the signer machine Sig_s from Definition 5.1 with the function $s := \mathsf{max\_skc}$ by letting $\mathsf{A}_{\mathsf{sig}}$ execute $\mathsf{C}_{\mathcal{H}}$, using the given A and H as blackboxes. It only has to choose an index $i \xleftarrow{\mathcal{R}} \{1, \ldots, n \cdot \mathsf{max\_in}(k)\}$ indicating for which of the up to $n \cdot \mathsf{max\_in}(k)$ key pairs generated due to inputs at ports $\mathsf{in}_u?$ with $u \in \mathcal{H}$ it uses pk obtained from the signer machine Sig_s instead. We showed above that the use of sk can be perfectly rewritten by interaction with Sig_s, and that $\mathsf{A}_{\mathsf{sig}}$ obtains a valid forgery if its guess i was correct.

Hence the success probability of $\mathsf{A}_{\mathsf{sig}}$ for each k is at least $(n \cdot \mathsf{max\_in}(k))^{-1}$ (from guessing i correctly) times the probability of $Forge_k$. Hence the security of the signature scheme implies that the probability of the sets $Forge_k$ is negligible.

8 Conclusion

We have presented a faithful abstraction of a crypto-library, which comprises important cryptographic operations like public-key encryption, digital signatures, and nonces. Faithful means that we can implement the abstraction securely in the cryptographic sense, so that properties proved for the abstraction carry over to the implementation without any further work. We provided one possible implementation whose security is based on provably secure cryptographic systems.

This faithfulness now allows for a meaningful analysis of protocol properties on the abstract level. As such an analysis is in scope of formal verification techniques, this bridges the up-to-now missing link between cryptography and formal methods. Exploring the complexity of our abstraction and further reducing it by developing or adapting currently used data-independence techniques is of independent interest.

As many protocols commonly analyzed in the literature can be expressed with our library, this enables the first formal, machine-aided verification of these protocols which is not only meaningful for Dolev-Yao-like abstractions, but whose security guarantees are equivalent to the security of the underlying cryptography.

Acknowledgments

We thank *Andre Adelsbach*, *Matthias Schunter*, and *Michael Steiner* for interesting discussions.

This work was partially supported by the European IST Project MAFTIA. However, it represents the view of the authors. The MAFTIA project is partially funded by the European Commission and the Swiss Department for Education and Science.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] 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.

[3] 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.

[4] R. Anderson and R. Needham. Robustness principles for public key protocols. In *Advances in Cryptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 1995.

[5] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, 2003. To appear.

[6] M. Backes, C. Jacobi, and B. Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Proc. 11th Symposium on Formal Methods Europe (FME 2002)*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2002.

[7] D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.

[8] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[11] R. Cramer and I. Damgård. Secure signature schemes based on interactive protocols. In *Advances in Cryptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 297–310. Springer, 1995.

[12] R. Cramer and I. Damgård. New generation of secure and practical RSA-based signatures. In *Advances in Cryptology: CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 173–185. Springer, 1996.

[13] R. Cramer and V. Shoup. Practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1998.

[14] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. In *Proc. 6th ACM Conference on Computer and Communications Security*, pages 46–51, 1999.

[15] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

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

[17] R. Gennaro, S. Halevi, and T. Rubin. Secure hash-and-sign signatures without the random oracle. In *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 1999.

[18] O. Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Advances in Cryptology: CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110. Springer, 1986.

[19] S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.

[20] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[21] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[22] J. D. Guttman, F. J. Thayer Fabrega, and L. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 186–195, 2001.

[23] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[24] R. Kemmerer. Analyzing encryption protocols using formal verfication techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.

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

[26] 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.

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

[28] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.

[29] S. Micali and P. Rogaway. Secure computation. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404. Springer, 1991.

[30] J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.

[31] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *Proc. 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. springer, 1992.

[32] B. Pfitzmann, M. Schunter, and M. Waidner. Secure reactive systems. Research Report RZ 3206, IBM Research, 2000.

[33] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254, 2000.

[34] 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.

[35] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer, 1992.

[36] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th Symposium on Principles of Programming Languages (POPL)*, pages 268–276, 2000.

[37] A. C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

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