# Symmetric Authentication
# Within a Simulatable Cryptographic Library


Michael Backes, Birgit Pfitzmann, Michael Waidner
IBM Zurich Research Lab
{mbc,bpf,wmi}@zurich.ibm.com



**Abstract**

Proofs of security protocols typically employ simple abstractions of cryptographic operations, so that large parts of such proofs are independent of cryptographic details. The typical abstraction is the Dolev-Yao model, which treats cryptographic operations as a specific term algebra. However, there is no cryptographic semantics, i.e., no theorem that says what a proof with the Dolev-Yao abstraction implies for the real protocol, even if provably secure cryptographic primitives are used.

Recently we introduced an extension to the Dolev-Yao model for which such a cryptographic semantics exists, i.e., where security is preserved if the abstractions are instantiated with provably secure cryptographic primitives. Only asymmetric operations (digital signatures and public-key encryption) are considered so far. Here we extend this model to include a first symmetric primitive, message authentication, and prove that the extended model still has all desired properties. The proof is a combination of a probabilistic, imperfect bisimulation with cryptographic reductions and static information-flow analysis.

Considering symmetric primitives adds a major complication to the original model: we must deal with the exchange of secret keys, which might happen any time before or after the keys have been used for the first time. Without symmetric primitives only public keys need to be exchanged.


## 1 Introduction

Proofs of security protocols typically employ simple abstractions of cryptographic operations, so that large parts of such proofs are independent of cryptographic details, such as polynomial-time restrictions, probabilistic behavior and error probabilities. This is particularly true for tool-supported proofs, e.g., [16, 15, 12, 21, 22, 1, 14, 17].

The typical abstraction is the Dolev-Yao model [8]: Cryptographic operations, e.g., $\mathsf{E}$ for encryption and $\mathsf{D}$ for decryption, are considered as operators in a term algebra where only certain cancellation rules hold. (In other words, one considers the initial model of an equational specification.) 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 $m$.

However, there is no cryptographic semantics, i.e., no theorem that says what a proof with the Dolev-Yao abstraction implies for the real protocol, even if provably secure cryptographic primitives are used. In fact, one can construct protocols that are secure in the Dolev-Yao model, but become insecure if implemented with certain provably secure cryptographic primitives [18]. Closing this gap has motivated a considerable amount of research in security and cryptography over the past few years, e.g., [13, 19, 3, 11, 20, 7, 4].

The abstraction we introduced in [4] achieved a major step towards closing this gap: We defined an ideal "cryptographic library" that offers abstract commands for generating nonces and keys, for performing operations (signing, testing, encrypting, decrypting) with these keys on messages, for dealing with lists and arbitrary application data, and for sending and receiving messages. The library further supports nested operations in the intuitive sense. The ideal cryptographic library has a simple deterministic behavior, and cryptographic objects are hidden at the interface, which makes it suitable as a basis for formal protocol verification. While the original Dolev-Yao model was a pure, memory-less algebra, our model is stateful, e.g., to distinguish different nonces and to reflect that cryptographically secure encryption and signature schemes are typically probabilistic. Thus our ideal cryptographic library corresponds more to "the CSP Dolev-Yao model" or "the Strand-space Dolev-Yao model" than the pure algebraic Dolev-Yao model.

This ideal cryptographic library is implemented by a real cryptographic library where the commands are implemented by real cryptographic algorithms and messages are actually sent between machines. The real system can be based on any cryptographically secure primitives. Our definition of security is based on the simulatability approach: security essentially means that anything an adversary against the real system can achieve can also be achieved by an adversary against the ideal system. This is the strongest possible cryptographic relation between a real and an ideal system. The definition in particular covers active attacks. In [4], our ideal and real cryptographic libraries were shown to fulfill this definition. The general composition theorem for the underlying model [20] implies that if a system that uses the abstract, ideal cryptographic library is secure then the same system using the real cryptographic library is also secure.

Only asymmetric cryptographic primitives (public-key encryption, digital signatures) are considered in [4], i.e., all primitives based on shared secret keys were not included. The main contribution of this paper is to add an important symmetric primitive to the framework of [4]: message authentication. We present abstractions for commands and data related to message authentication, e.g., commands for key generation, authentication, and authenticator testing, and we present a concrete realization based on cryptographic primitives. We then show that these two systems fulfill the simulatability definition if they are plugged into the existing cryptographic library. The inclusion of symmetric primitives and the sending of secret keys add a major complication to the original proof, because keys may be sent any time before or after the keys have been used for the first time. In particular, this implies that a real adversary can send messages which cannot immediately be simulated by a known term, because the keys needed to test the validity of authenticators are not yet known, but may be sent later by the adversary. Without symmetric primitives only public keys had to be exchanged, and the problem could be avoided by appropriate tagging of all real messages with the public keys used in them, so that messages could be immediately classified into correct terms or a specific garbage type [4].

**Related Work.** Abadi et. al. [3, 2] started to bridge the abstraction gap by considering a Dolev-Yao model with nested algorithms specifically for symmetric encryption and synchronous protocols. 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 were all formulated as indistinguishability, i.e., if two abstract systems are indistinguishable by passive adversaries, then this is also true for the two corresponding real systems. This model does not yet contain theorems about composition or property preservation from the abstract to the real system. The price we pay for the greater applicability of reactive simulatability and for allowing active attacks is a much more complicated proof.

Several papers extended this work for specific models of specific classes of protocols. For instance, [10] specifically considers strand spaces, and within this model information-theoretically secure primitives.

The recent definitions of simulatability for reactive systems come with more or less worked-out examples of abstractions of cryptographic systems; however, even with a composition theorem this does not automatically give a cryptographic library in the Dolev-Yao sense, i.e., with the possibility to nest abstract operations. For instance, the abstract secure channels in [20] combine encryption and signatures in a fixed way, while the lower-level encryption subsystem used in that paper, like the examples in [13], does not offer abstract, implementation-independent outputs. The cryptographic primitives in [6, 7] are abstract, but do not support nested operations: ideal cryptographic operations are defined through immediate interactions with the adversary, i.e., they are not local to the party that performs them and the adversary learns the structure of every term any honest party ever builds. The ideal system for signatures even reveals every signed message to the adversary. Thus, by composing cryptographic operations already the ideal systems reveal too much information to the adversary; thus they cannot be a sound basis for more complex protocols.

Our cryptographic library overcomes these problems. It supports nested operations in the intuitive sense; operations that are performed locally are not visible to the adversary. It is secure against arbitrary active attacks, and the composition theorem allows for safely using it within the context of arbitrary surrounding interactive protocols. This holds independently of the goals that one wants to prove about the surrounding protocols.

## 2 Overview of Simulatability

We start with a brief overview of the underlying security notion of simulatability, which is the basic notion for comparing an ideal and a real system. For the moment, we only need to know that an ideal and a real system each consist of several possible structures, typically derived from an intended structure with a trust model. An intended structure represents a benign world, where each user is honest and each machine behaves as specified. The trust model is then used to determine the potentially malicious machines, i.e., machines which are considered to be under control of the adversary. Moreover, the trust model classifies the "security" of each connection between machines of the structure, e.g., that a connection is authentic, but not secret. Now for each element of the trust model, this gives one separate structure.

Each structure interacts with an adversary $\mathsf{A}$ and honest users summarized as a single machine $\mathsf{H}$. The security definition is that for all polynomial-time honest users $\mathsf{H}$ and all polynomial-time adversaries $\mathsf{A}_1$ on the real system, there exists an adversary $\mathsf{A}_2$ on the ideal system such that the honest users $\mathsf{H}$ cannot notice the difference, as shown in Figure 1.

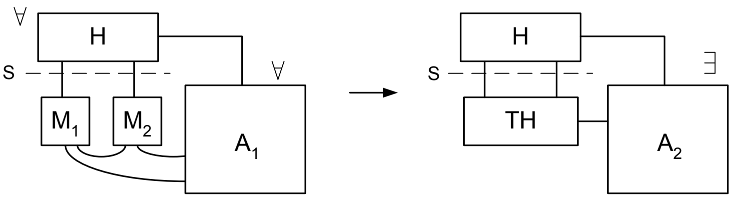


Figure 1: Overview of the simulatability definition. A real system is shown on the left-hand side, and an ideal system on the right-hand side. The view of $\mathsf{H}$ must be indistinguishable.

## 3 The Ideal System

For modeling and proving cryptographic protocols using our abstraction, it is sufficient to understand and use the ideal cryptographic library described in this section. Thus, applying our results to the verification of cryptographic protocols does not presuppose specific knowledge about cryptography or probabilism. The subsequent sections only justify the cryptographic faithfulness of this ideal library.

The ideal cryptographic 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 behavior in the ideal system. 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 of the database has a type (e.g., "signature"), and pointers to its arguments (e.g., a key and a message). This corresponds to the top level of a Dolev-Yao term; an entire term can be found by following the pointers. Further, each entry contains handles for those participants who already know it. The reason for using handles to make an entry accessible for higher protocols is that an idealized cryptographic term and the corresponding real message have to be presented in the same way to higher protocols to allow for a provably secure implementation in the sense of simulatability. In the ideal library, handles essentially point to Dolev-Yao-like terms, while in the real library they point to cryptographic messages.

The ideal cryptographic library does not allow cheating by construction. For instance, if it receives a command to encrypt a message $m$ with a certain key, it simply makes an abstract database entry for the ciphertext. Another user can only ask for decryption of this ciphertext if he has handles to both the ciphertext and the secret key. Similarly, if a user issues a command to sign a message, the ideal system looks up whether this user should have the secret key. If yes, it stores that this message has been signed with this key. Later tests are simply look-ups in this database. A send operation makes an entry known to other participants, i.e., it adds handles to the entry. Our model does not only cover crypto operations, but it is an entire reactive system and therefore contains an abstract network model.

In the following, we present our additions to this ideal system for capturing symmetric authentication primitives, i.e., providing abstractions from authentication keys and authenticators, and offering commands for key generation, authentication, and verification. Both authenticators and authentication keys can be included into messages that are sent over the network, which allows for sharing authentication keys with other participants. Before we introduce our additions in detail, we explain the major design decisions. For understanding these decision, it might be helpful for readers not familiar with message authentication to read Section 4.1 before, which contains the cryptographic definition of secure authentication schemes.

First, we have to allow for checking if authenticators have been created with the same secret key; as the definition of secure authentication schemes does not exclude this, it can happen in the real system. For public-key encryption and digital signatures, this was achieved in [4] by tagging ciphertexts respectively signatures with the corresponding public key, so that the public keys can be compared. For authenticators, this is clearly not possible as no public key exists there. We solve this problem by tagging authenticators with an "empty" public key, which serves as a key identifier for the secret key.

Secondly, as authentication keys can be exchanged between the users and the adversary, it might happen that an authenticator is valid with respect to several authentication keys, e.g., because the adversary has created a suitable key. Hence it must be possible to tag authenticators with additional key identifiers during the execution, i.e., authenticators are tagged with a list of key identifiers. This list can also be empty which models authenticators from the adversary for which no suitable key is known yet.

Thirdly, we have to reflect the special capabilities an adversary has in the real system. For example, he might be able to transform an authenticator, i.e., to create a new authenticator for a message for which the correct user has already created another authenticator. Such a transformation is not excluded in the definition of secure authentication schemes, hence it might happen in the real system. The ideal library therefore offers special commands for the adversary to model capabilities of this kind.

### 3.1 Notation

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

### 3.2 Structures and Parameters

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

The ideal system keeps track of the length of messages using a tuple $L$ of abstract length functions. We add functions $\mathsf{ska\_len}^*(k)$ and $\mathsf{aut\_len}^*(k, l)$ to $L$ for the length of authentication keys and authenticators, depending on a security parameter $k$ and the length $l$ of the message, with the same conventions as for the other functions in [4]. In particular, this means that they range over $\mathbb{N}$, are polynomially bounded, and efficiently computable.

### 3.3 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 $\mathsf{p}?$. The database $D$ contains abstractions from real cryptographic objects which correspond to the top levels of Dolev-Yao terms. An entry has the following attributes:

- $x.ind \in \mathcal{INDS}$, called index, consecutively numbers all entries in $D$. The set $\mathcal{INDS}$ is isomorphic to $\mathbb{N}$; we use it to distinguish index arguments from others. We use the index as a primary key attribute of the database, i.e., we write $D[i]$ for the selection $D[ind = i]$.

- $x.type \in typeset$ identifies the type of $x$. We add types $\mathsf{ska}$, $\mathsf{pka}$, and $\mathsf{aut}$ to *typeset* from [4], denoting secret authentication keys, "empty" public keys that are needed as key identifier for the corresponding authentication keys, and authenticators.

- $x.arg = (a_1, a_2, \ldots, a_j)$ is a possibly empty list of arguments. Many values $a_i$ are indices of other entries in $D$ and thus in $\mathcal{INDS}$. We sometimes distinguish them by a superscript "ind".

- $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{\mathsf{a}\}$ are handles by which a user or adversary $u$ knows this entry. $x.hnd_u = \downarrow$ means that $u$ does not know this entry. The set $\mathcal{HNDS}$ is yet another set isomorphic to $\mathbb{N}$. We always use a superscript "hnd" for handles.

- $x.len \in \mathbb{N}_0$ denotes the "length" of the entry, which is computed by applying the functions from $L$.

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

For each input port $\mathsf{p}?$, $\mathsf{TH}_\mathcal{H}$ maintains a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ initialized with 0 for the number of inputs at that port, each with a bound $bound_{\mathsf{p}?}$. If that bound is reached, no further inputs are accepted at that port, which is used to achieve polynomial runtime of the machine $\mathsf{TH}_\mathcal{H}$ independent of the environment. This is done by a length function becoming 0; these length functions can generally be used to ensure that only polynomial-size inputs are considered at certain ports. They are not written out explicitly, but can be derived easily from the domain expectations given for the individual inputs. We have $bound_{\mathsf{p}?} = \mathsf{max\_in}(k)$ for all ports except for $\mathsf{in_a}?$, where it can be a specific bound $\mathsf{max\_in_a}(k)$ or any larger polynomial. Here, $\mathsf{max\_in}$ is a parameter of the cryptographic library and can be an arbitrary polynomial; it denotes the maximum number of inputs at each user ports. In contrast, $\mathsf{max\_in_a}$ is derived in [4] according to the security proof. Intuitively, it ensures polynomial runtime, but is still large enough to ensure correct functional behavior, since it is never reached in a simulation. We have to enlarge $\mathsf{max\_in_a}$ in this work to allow for a correct simulation, but since any larger polynomial will do, our security proof only has to ensure that our new bound is indeed polynomially bounded.

### 3.4 New Inputs and their Evaluation

The ideal system has several types of inputs: Basic commands are accepted at all ports $\mathsf{in}_u?$; they correspond to cryptographic operations and have only local effects, i.e., only an output at the port $\mathsf{out}_u?$ for the same user occurs and only handles for $u$ are involved. Local adversary commands are of the same type, but only accepted at $\mathsf{in_a}?$; they model tolerated imperfections, i.e., possibilities that an adversary may have, but honest users do not. Send commands output values to other users. In the following, the notation $j \leftarrow \mathsf{algo}(i)$ for a command $\mathsf{algo}$ of $\mathsf{TH}_\mathcal{H}$ means that $\mathsf{TH}_\mathcal{H}$ receives an input $\mathsf{algo}(i)$ and outputs $j$ if the input and output port are clear from the context. We only allow lists to be authenticated and transferred, because the list-operation is a convenient place to concentrate all verifications that no secret keys of the public-key systems from [4] are put into messages.

For dealing with symmetric authentication we only have to add new basic commands and local adversary commands; the send commands are unchanged. We now define the precise new inputs and how $\mathsf{TH}_\mathcal{H}$ evaluates them. Handle arguments are tacitly required to be in $\mathcal{HNDS}$ and existing, i.e., $\leq curhnd_u$, at the time of execution.

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

### 3.4.1 Basic Commands

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

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

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

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

- *Authenticator generation:* $aut^{\mathsf{hnd}} \leftarrow \mathsf{auth}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

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

$$\begin{aligned} D \;&:\Leftarrow\; (ind := size\texttt{++}, type := \mathsf{aut}, arg := (l, pka),\\ &\qquad hnd_u := aut^{\mathsf{hnd}}, len := length).\end{aligned}$$

  The general argument format for entries of type $\mathsf{aut}$ is $(l, pka_1, \ldots, pka_j)$. The arguments $pka_1, \ldots, pka_j$ are the key identifiers of those secret keys for which this authenticator is valid. We will see in Section 3.4.2 that additional key identifiers for an authenticator can be added during the execution, e.g., because the adversary has created a suitable key. Such arguments are appended at the end of the existing list. An empty sequence of arguments $pka_i$ models authenticators from the adversary for which no suitable secret key has been received yet.

- *Authenticator verification:* $v \leftarrow \mathsf{auth\_test}(aut^{\mathsf{hnd}}, ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  If $aut := D[hnd_u = aut^{\mathsf{hnd}} \wedge type = \mathsf{aut}].ind = \downarrow$ or $ska := D[hnd_u = ska^{\mathsf{hnd}} \wedge type = \mathsf{ska}].ind = \downarrow$, return $\downarrow$. Otherwise, let $(l, pka_1, \ldots, pka_j) := D[aut].arg$. If $ska - 1 \notin \{pka_1, \ldots, pka_j\}$ or $D[l].hnd_u \neq l^{\mathsf{hnd}}$, then $v := \mathsf{false}$, else $v := \mathsf{true}$.

  The test $ska - 1 \in \{pka_1, \ldots, pka_j\}$ is the lookup that the secret key is one of those for which this authenticator is valid, i.e., that the cryptographic test would be successful in the real system.

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

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

[1] This command implies that real authenticators must contain the message. The simulator in the proof needs this to translate authenticators from the adversary into abstract ones. Thus we also offer message retrieval to honest users so that they need not send the message separately.

### 3.4.2 Local Adversary Commands

The following local commands are only accepted at the port $\mathsf{in_a}?$. They model special capabilities of the adversary. This comprises *authentication transformation*, which allows the adversary to create a new authenticator for a message provided that the adversary already has a handle for another authenticator for the same message. This capability has to be included in order to be securely realizable by cryptographic primitives, since the security definition of authentication schemes does not exclude such a transformation.

If an authenticator is received from the adversary for which no suitable key has been received yet, we call the authenticator (temporarily) unknown. In the real system, this means that no user will be able to check the validity of the authenticator. In the ideal system, this is modeled by providing a command for generating an *unknown authenticator*. Such an authenticator can become valid if a suitable secret key is received. A command for *fixing authenticators* takes care of this. Finally, we allow the adversary to retrieve all information that we do not explicitly require to be hidden, e.g., arguments and the type of a given handle. This is dealt with by a command for *parameter retrieval*.

- *Authentication transformation:* $trans\_aut^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_aut}(aut^{\mathsf{hnd}})$.

  Return $\downarrow$ if $aut := D[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}} \wedge type = \mathsf{aut}].ind = \downarrow$. Otherwise let $(l, pka_1, \ldots, pka_j) := D[aut].arg$, set $trans\_aut^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{++}$ and

  $$\begin{aligned} D \;:\Leftarrow\; & (ind := size{++}, type := \mathsf{aut}, arg := (l, pka_1), \\ & hnd_{\mathsf{a}} := trans\_aut^{\mathsf{hnd}}, len := D[aut].len). \end{aligned}$$

- *Unknown authenticator:* $aut^{\mathsf{hnd}} \leftarrow \mathsf{adv\_unknown\_aut}(l^{\mathsf{hnd}})$.

  Return $\downarrow$ if $l := D[hnd_{\mathsf{a}} = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind = \downarrow$ or $length := \mathsf{aut\_len}^*(k, D[l].len) > \mathsf{max\_len}(k)$. Otherwise, set $aut^{\mathsf{hnd}} := curhnd_a{++}$ and

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

  Note that no key identifier exists for this authenticator yet.

- *Fixing authenticator:* $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, aut^{\mathsf{hnd}})$.

  Return $\downarrow$ if $aut := D[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}} \wedge type = \mathsf{aut}].ind = \downarrow$ or if $ska := D[hnd_u = ska^{\mathsf{hnd}} \wedge type = \mathsf{ska}].ind = \downarrow$. Let $(l, pka_1, \ldots, pka_j) := D[aut].arg$ and $pka := ska - 1$. If $pka \notin \{pka_1, \ldots, pka_j\}$ set $D[aut].arg := (l, pka_1, \ldots, pka_j, pka)$ and output $v := \mathsf{true}$. Otherwise, output $v := \mathsf{false}$.

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

  This existing command always sets $type := D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].type$, and for most types $arg := \mathsf{ind2hnd}_{\mathsf{a}}^*(D[hnd_{\mathsf{a}} = m^{\mathsf{hnd}}].arg)$. This applies to the new types $\mathsf{pka}$, $\mathsf{ska}$, and $\mathsf{aut}$.

Note that for authenticators, a handle to the "empty" public key is output in $\mathsf{adv\_parse}$. If the adversary wants to know whether two given authenticators have been created using the same secret key, it simply parses them yielding handles to the corresponding "empty" public keys, and compares these handles.

### 3.4.3 Send Commands

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

- $\mathsf{send\_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$. Intuitively, the list $l$ shall be sent to user $v$. Let $l^{\mathsf{ind}} := D[hnd_u = l^{\mathsf{hnd}} \wedge type = \mathsf{list}].ind$. If $l^{\mathsf{ind}} \neq \downarrow$, then output $(u, v, \mathsf{ind2hnd_a}(l^{\mathsf{ind}}))$ at $\mathsf{out_a}!$.
- $\mathsf{adv\_send\_i}(u, v, l^{\mathsf{hnd}})$, for $u \in \{1, \ldots, n\}$ 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 $l^{\mathsf{ind}} \neq \downarrow$ output $(u, v, \mathsf{ind2hnd}_v(l^{\mathsf{ind}}))$ at $\mathsf{out}_v!$.

## 3.5 Properties of the Ideal System

All properties shown about the ideal system in Lemmas 4.1 and 4.2 of [4] still hold, e.g., "well-defined terms" stating that the database $D$ represents well-defined, non-cyclic terms. The invariant "correct key pairs" is extended by $D[i].type = \mathsf{pka} \iff D[i+1].type = \mathsf{ska}$ for all $i \in \mathbb{N}_0$.

We recall the definition $\mathsf{owners}(x) := \{u \in \mathcal{H} \cup \{\mathsf{a}\} \mid x.hnd_u \neq \downarrow\}$ for $x \in D$. If $|\mathsf{owners}(x)| = 1$, we write $\mathsf{owner}(x)$ for the element of $\mathsf{owners}(x)$.

# 4 Real System

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

## 4.1 Cryptographic Operations

We denote a memoryless symmetric authentication scheme by a tuple $\mathcal{A} = (\mathsf{gen_A}, \mathsf{auth}, \mathsf{atest}, \mathsf{ska\_len}, \mathsf{aut\_len})$ of polynomial-time algorithms. For authentication key generation for a security parameter $k \in \mathbb{N}$, we write

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

The length of $sk$ is $\mathsf{ska\_len}(k) > 0$. By

$$aut \leftarrow \mathsf{auth}_{sk}(m)$$

we denote the (probabilistic) authentication of a message $m \in \{0,1\}^+$. Verification

$$b := \mathsf{atest}_{sk}(aut, m)$$

is deterministic and returns $\mathsf{true}$ (then we say that the authenticator is valid) or $\mathsf{false}$. Correctly generated authenticators for keys of the correct length must always be valid. The length of $aut$ is $\mathsf{aut\_len}(k, |m|) > 0$. This is also true for every $aut'$ with $\mathsf{atest}_{sk}(aut', m) = \mathsf{true}$ for a value $sk \in \{0,1\}^{\mathsf{ska\_len}(k)}$. The functions $\mathsf{ska\_len}$ and $\mathsf{aut\_len}$ must be bounded by multivariate polynomials.

As the security definition we use security against existential forgery under adaptive chosen-message attacks similar to [9]. We only use our notation for interacting machines, and we allow that also the test function is adaptively attacked.

**Definition 4.1** *(Authentication Security) Given an authentication scheme, an* authentication machine $\mathsf{Aut}$ *has one input and one output port, a variable sk initialized with* $\downarrow$*, and the following transition rules:*

- *First generate a key as* $sk \leftarrow \mathsf{gen_A}(1^k)$.
- *On input* $(\mathsf{auth}, m_j)$*, return* $aut_j \leftarrow \mathsf{auth}_{sk}(m_j)$.
- *On input* $(\mathsf{test}, aut', m')$*, return* $v := \mathsf{atest}_{sk}(aut', m')$.

*The authentication scheme is called* existentially unforgeable under adaptive chosen-message attack *if for every probabilistic polynomial-time machine* $\mathsf{A_{aut}}$ *that interacts with* $\mathsf{Aut}$*, the probability is negligible (in k) that* $\mathsf{Aut}$ *outputs* $v = \mathsf{true}$ *on any input* $(\mathsf{test}, aut', m')$ *where* $m'$ *was not authenticated until that time, i.e., not among the* $m_j$*'s.* ◇

Note that the definition does not exclude authenticator transformation, i.e., if a message $m_i$ has been properly authenticated, creating another valid authenticator for $m_i$ is not excluded. This is why we introduced the command $\mathsf{adv\_transform\_aut}$ as a tolerable imperfection in Section 3.4.2. A well-known example of an authentication scheme that is provably secure under this definition is HMAC [5].

## 4.2 Structures

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

## 4.3 Lengths and Bounds

In the real system, we have length functions $\mathsf{list\_len}$, $\mathsf{nonce\_len}$, $\mathsf{ska\_len}$, and $\mathsf{aut\_len}$, corresponding to the length of lists, nonces, authentication keys, and authenticators, respectively. These functions can be arbitrary polynomials. For given functions $\mathsf{list\_len}$, $\mathsf{nonce\_len}$, $\mathsf{ska\_len}$, and $\mathsf{aut\_len}$, the corresponding ideal length functions are computed as follows:

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

### 4.4 States of a Machine

The state of each machine $\mathsf{M}_u$ consists of a database $D_u$ and variables $curhnd_u$ and $steps_{\mathsf{p}?}$ for each input port $\mathsf{p}?$. Each entry $x$ in $D_u$ has the following attributes:

- $x.hnd_u \in \mathcal{HNDS}$ consecutively numbers all entries in $D_u$. We use it as a primary key attribute, i.e., we write $D_u[i^{\mathsf{hnd}}]$ for the selection $D_u[hnd_u = i^{\mathsf{hnd}}]$.
- $x.word \in \{0, 1\}^+$ is the real representation of $x$.
- $x.type \in typeset \cup \{\mathsf{null}\}$ identifies the type of $x$. The value $\mathsf{null}$ denotes that the entry has not yet been parsed.
- $x.add\_arg$ is a list of ("additional") arguments. For entries of our new types it is always $()$.

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

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

determines a handle for certain given parameters in $D_u$: If an entry with the word $i$ already exists, i.e., $i^{\mathsf{hnd}} := D_u[word = i \wedge type \notin \{\mathsf{sks}, \mathsf{ske}\}].hnd_u \neq \downarrow$,[2] it returns $i^{\mathsf{hnd}}$, assigning the input values $type$ and $add\_arg$ to the corresponding attributes of $D_u[i^{\mathsf{hnd}}]$ only if $D_u[i^{\mathsf{hnd}}].type$ was $\mathsf{null}$. Else if $|i| > \mathsf{max\_len}(k)$, it returns $i^{\mathsf{hnd}} = \downarrow$. Otherwise, it sets and returns $i^{\mathsf{hnd}} := curhnd_u{+}{+}$, $D_u :\Leftarrow (i^{\mathsf{hnd}}, i, type, add\_arg)$.

For each input port $\mathsf{p}?$, $\mathsf{M}_u$ 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 $\mathsf{max\_in}(k)$. Length functions for inputs are tacitly defined by the domains of each input again.

### 4.5 Inputs and their Evaluation

Now we describe how $\mathsf{M}_u$ evaluates individual new inputs.

#### 4.5.1 Constructors and One-level Parsing

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

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

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

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

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

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

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

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

### 4.5.2 Basic Commands and $\mathsf{parse\_}\mathit{type}$

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

- *Key constructor:* $\mathit{sk}^* \leftarrow \mathsf{make\_auth\_key}()$.

  Let $\mathit{sk} \leftarrow \mathsf{gen_A}(1^k)$, $\mathit{sr} \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, and return $\mathit{sk}^* := (\mathsf{ska}, \mathit{sk}, \mathit{sr})$.

- *Key generation:* $\mathit{ska}^{\mathsf{hnd}} \leftarrow \mathsf{gen\_auth\_key}()$.

  Let $\mathit{sk}^* \leftarrow \mathsf{make\_auth\_key}()$, $\mathit{ska}^{\mathsf{hnd}} := \mathit{curhnd}_u\mathsf{++}$, and $D_u :\Leftarrow (\mathit{ska}^{\mathsf{hnd}}, \mathit{sk}^*, \mathsf{ska}, ())$.

- *Key parsing:* $\mathit{arg} \leftarrow \mathsf{parse\_ska}(\mathit{sk}^*)$.

  If $\mathit{sk}^*$ is of the form $(\mathsf{ska}, \mathit{sk}, \mathit{sr})$ with $\mathit{sk} \in \{0,1\}^{\mathsf{ska\_len}(k)}$ and $\mathit{sr} \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, return $()$, else $\downarrow$.

- *Authenticator constructor:* $\mathit{aut}^* \leftarrow \mathsf{make\_auth}(\mathit{sk}^*, l)$, for $\mathit{sk}^*, l \in \{0,1\}^+$.

  Set $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$, $\mathit{sk} := \mathit{sk}^*[2]$ and $\mathit{sr} := \mathit{sk}^*[3]$. Authenticate as $\mathit{aut} \leftarrow \mathsf{auth}_{\mathit{sk}}((r, l))$, and return $\mathit{aut}^* := (\mathsf{aut}, \mathit{sr}, r, l, \mathit{aut})$.

- *Authenticator generation:* $\mathit{aut}^{\mathsf{hnd}} \leftarrow \mathsf{auth}(\mathit{ska}^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Parse $l^{\mathsf{hnd}}$ if necessary. If $D_u[\mathit{ska}^{\mathsf{hnd}}].\mathit{type} \neq \mathsf{ska}$ or $D_u[l^{\mathsf{hnd}}].\mathit{type} \neq \mathsf{list}$, then return $\downarrow$. Otherwise set $\mathit{sk}^* := D_u[\mathit{ska}^{\mathsf{hnd}}].\mathit{word}$, $l := D_u[l^{\mathsf{hnd}}].\mathit{word}$, and $\mathit{aut}^* \leftarrow \mathsf{make\_auth}(\mathit{sk}^*, l)$. If $|\mathit{aut}^*| > \mathsf{max\_len}(k)$, return $\downarrow$, else set $\mathit{aut}^{\mathsf{hnd}} := \mathit{curhnd}_u\mathsf{++}$ and $D_u :\Leftarrow (\mathit{aut}^{\mathsf{hnd}}, \mathit{aut}^*, \mathsf{aut}, ())$.

- *Authenticator parsing:* $\mathit{arg} \leftarrow \mathsf{parse\_aut}(\mathit{aut}^*)$.

  If $\mathit{aut}^*$ is not of the form $(\mathsf{aut}, \mathit{sr}, r, l, \mathit{aut})$ with $\mathit{sr}, r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, $l \in \{0,1\}^+$, and $\mathit{aut} \in \{0,1\}^{\mathsf{aut\_len}'(k,|l|)}$, return $\downarrow$. Also return $\downarrow$ if $l$ is not a tagged list. Otherwise set $\mathit{arg} := (l)$.

- *Authenticator verification:* $v \leftarrow \mathsf{auth\_test}(\mathit{aut}^{\mathsf{hnd}}, \mathit{ska}^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Parse $\mathit{aut}^{\mathsf{hnd}}$ yielding $\mathit{arg} =: (l)$, and parse $\mathit{ska}^{\mathsf{hnd}}$. If $D_u[\mathit{aut}^{\mathsf{hnd}}].\mathit{type} \neq \mathsf{aut}$ or $D_u[\mathit{ska}^{\mathsf{hnd}}].\mathit{type} \neq \mathsf{ska}$, return $\downarrow$. Else let $(\mathsf{aut}, \mathit{sr}, r, l, \mathit{aut}) := D_u[\mathit{aut}^{\mathsf{hnd}}].\mathit{word}$ and $\mathit{sk} := D_u[\mathit{ska}^{\mathsf{hnd}}].\mathit{word}[2]$. If $\mathit{sr} \neq D_u[\mathit{ska}^{\mathsf{hnd}}].\mathit{word}[3]$

  or $l \neq D_u[l^{\mathsf{hnd}}].\mathit{word}$, or $\mathsf{atest}_{\mathit{sk}}(\mathit{aut}, (r, l)) = \mathsf{false}$, output $v := \mathsf{false}$, else $v := \mathsf{true}$.

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

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

### 4.5.3 Send Commands and Network Inputs

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

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

# 5 Simulator

We now start with the proof that the real system is as secure as the ideal one. The main step is to construct a simulator $\mathsf{Sim}_{\mathcal{H}}$ for each set $\mathcal{H}$ of possible honest users such that for every real adversary $\mathsf{A}$, the combination $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ of $\mathsf{Sim}_{\mathcal{H}}$ and $\mathsf{A}$ achieves the same effects in the ideal system as the adversary $\mathsf{A}$ in the real system, cf. Section 2. This is shown in Figure 2. This figure also shows the ports of $\mathsf{Sim}_{\mathcal{H}}$. Roughly, the goal of $\mathsf{Sim}_{\mathcal{H}}$ is to translate real bitstrings coming from the adversary into abstract handles that represent corresponding terms in $\mathsf{TH}_{\mathcal{H}}$, and vice versa. This will be described in the following.

## 5.1 States of the Simulator

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

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

The variable $curhnd_{\mathsf{a}}$ denotes the current size of $D_{\mathsf{a}}$, except temporarily within an algorithm $\mathsf{id2real}$. The variables $steps_{\mathsf{p}?}$ count the inputs at each port. The corresponding bounds $bound_{\mathsf{p}?}$ are $\mathsf{max\_in}(k)$ for the network ports and $\mathsf{max\_in_a}(k)$ for $\mathsf{out_a}?$. (These bounds were introduced in Section 3.3.)

## 5.2 Input Evaluation of Send Commands

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

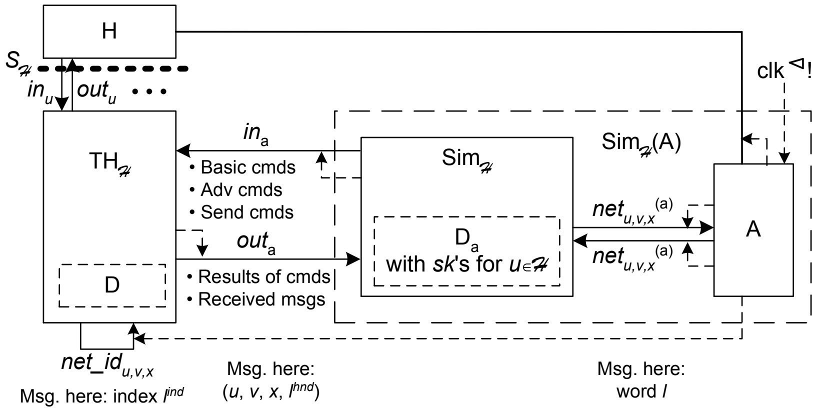


Figure 2: Set-up of the simulator.

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

1. Call $(type, (m_1^{\mathsf{hnd}}, \ldots, m_j^{\mathsf{hnd}})) \leftarrow \mathsf{adv\_parse}(m^{\mathsf{hnd}})$ at $\mathsf{in_a}!$, expecting $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ and $j \leq \mathsf{max\_len}(k)$, and $m_i^{\mathsf{hnd}} \leq \mathsf{max\_hnd}(k)$ if $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and otherwise $|m_i^{\mathsf{hnd}}| \leq \mathsf{max\_len}(k)$ (with certain domain expectations in the arguments $m_i^{\mathsf{hnd}}$ that are automatically fulfilled in interaction with $\mathsf{TH}_{\mathcal{H}}$, also for the now extended command adv_parse for the new types).

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

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

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

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

### 5.3 Evaluation of Network Inputs

When $\mathsf{Sim}_{\mathcal{H}}$ receives a bitstring $l$ from A at a port $\mathsf{net}_{w,u,\mathsf{i}}?$ with $|l| \leq \mathsf{max\_len}(k)$, it verifies that $l$ is a tagged list. If yes, it translates $l$ into a corresponding handle $l^{\mathsf{hnd}}$ and outputs the abstract sending command $\mathsf{adv\_send\_i}(w, u, l^{\mathsf{hnd}})$ at port $\mathsf{in_a}!$.

For an arbitrary message $m \in \{0,1\}^+$, $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$ works as follows. If there is already a handle $m^{\mathsf{hnd}}$ with $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, then real2id reuses that. Otherwise it recursively parses the real message, builds up a corresponding term in $\mathsf{TH}_{\mathcal{H}}$, and enters all messages into $D_{\mathsf{a}}$. For building up the abstract term, real2id makes extensive use of the special adversary capabilities that $\mathsf{TH}_{\mathcal{H}}$ provides. In the real system, the bitstring may, e.g., contain an authenticator for which no matching authentication key is known yet. Therefore, the simulator has to be able to insert such an authenticator with "unknown" key into the database of $\mathsf{TH}_{\mathcal{H}}$, which explains the need for the command adv_unknown_aut. Similarly, the adversary might send a new authentication key, which has to be added to all existing authenticator entries for which this key is valid, or he might send a transformed authenticator, i.e., a new authenticator for a message for which the correct user has already created another authenticator. Such a transformation is not excluded by the definition of secure authentication schemes, hence it might occur in the real system. All these cases can be covered by using the special adversary capabilities.

Formally, id2real sets $(type, arg) := \mathsf{parse}(m)$ and calls a type-specific algorithm $add\_arg \leftarrow \mathsf{real2id\_}type(m, arg)$. After this, real2id sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}\texttt{++}$ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$. We have to provide the type-specific algorithms for our new types.

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

  Let $m =: (\mathsf{ska}, sk, sr)$; this format is ensured by the preceding parsing. For each handle $aut^{\mathsf{hnd}}$ with $D_{\mathsf{a}}[aut^{\mathsf{hnd}}].type = \mathsf{aut}$ and $D_{\mathsf{a}}[aut^{\mathsf{hnd}}].word = (\mathsf{aut}, sr, r, l, aut)$ for $r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, $l \in \{0,1\}^+$, and $aut \in \{0,1\}^{\mathsf{aut\_len}'(k,|l|)}$, and $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$, call $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, aut^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Return $add\_arg$.

- $add\_arg \leftarrow \mathsf{real2id\_aut}(m, (l))$. Make a recursive call $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ and let $(\mathsf{aut}, sr, r, l, aut) := m$; parsing ensures this format.

  Let $Ska := \{ska^{\mathsf{hnd}} \mid D_{\mathsf{a}}[ska^{\mathsf{hnd}}].type = \mathsf{ska} \wedge D_{\mathsf{a}}[ska^{\mathsf{hnd}}].word[3] = sr \wedge \mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$ for $sk := D_{\mathsf{a}}[ska^{\mathsf{hnd}}].word[2]\}$ be the set of keys known to the adversary for which $m$ is valid.

  Verify whether the adversary has already seen another authenticator for the same message with a key only known to honest users:

  Let $Aut := \{aut^{\mathsf{hnd}} \mid D_{\mathsf{a}}[aut^{\mathsf{hnd}}].word = (\mathsf{aut}, sr, r, l, aut') \wedge D_{\mathsf{a}}[aut^{\mathsf{hnd}}].type = \mathsf{aut}\}$. For each $aut^{\mathsf{hnd}} \in Aut$, let $(\mathsf{aut}, arg_{aut^{\mathsf{hnd}}}) \leftarrow \mathsf{adv\_parse}(aut^{\mathsf{hnd}})$ and $pka_{aut^{\mathsf{hnd}}} := arg_{aut^{\mathsf{hnd}}}[2]$.

  We claim that there exists at most one $pka_{aut^{\mathsf{hnd}}}$ such that the corresponding secret key was generated by an honest user, i.e., such that $D_{\mathsf{a}}[pka_{aut^{\mathsf{hnd}}}].add\_arg[1] = \mathsf{honest}$. If such a $pka_{aut^{\mathsf{hnd}}}$ exists, let $sk^* := D_{\mathsf{a}}[pka_{aut^{\mathsf{hnd}}}].add\_arg[2]$ and $v := \mathsf{atest}_{sk^*[2]}(aut, (r, l))$. If $v = \mathsf{true}$, call $trans\_aut^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_aut}(aut^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and after that call $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, trans\_aut^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ for every $ska^{\mathsf{hnd}} \in Ska$. Return $()$.

  Else if $Ska \neq \emptyset$, let $ska^{\mathsf{hnd}} \in Ska$ arbitrary. Call $aut^{\mathsf{hnd}} \leftarrow \mathsf{auth}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$, and for every $ska'^{\mathsf{hnd}} \in Ska \setminus \{ska^{\mathsf{hnd}}\}$ (in any order), call $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska'^{\mathsf{hnd}}, aut^{\mathsf{hnd}})$ at $\mathsf{in_a}!$. Return $()$.

  If $Ska = \emptyset$, call $aut^{\mathsf{hnd}} \leftarrow \mathsf{adv\_unknown\_aut}(l^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and return $()$.

## 5.4 Properties of the Simulator

Two important properties have to be shown for the simulator. First, it has to be polynomial-time, as the joint adversary $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ might otherwise not be a valid polynomial-time adversary on the ideal system. Secondly, we have to show that the interaction between $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ in the recursive algorithms cannot fail because one of the machine reaches its runtime bound.

Essentially, this can be shown as in [4], except that the interaction of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ in real2id can additionally increase the number of steps linearly in the number of existing authenticators and existing keys, since a new secret key might update the arguments of each existing authenticator entry, and a new authenticator can get any existing key as an argument. This is the reason why we had to enlarge the original bound $\mathsf{max\_in}_\mathsf{a}$ at $\mathsf{in}_\mathsf{a}?$ and $\mathsf{out}_\mathsf{a}?$ to maintain the correct functionality of the simulator, cf. Section 3.3 and 5.1. However, only a polynomial number of authenticators and keys can be created (a coarse bound is $n \cdot \mathsf{max\_in}(k)$ for entries generated by honest users plus the polynomial runtime of A for the remaining ones). We omit further details.

# 6 Proof of Correct Simulation

Given the simulator, we show that even the combination of arbitrary polynomial-time users H and an arbitrary polynomial-time adversary A cannot distinguish the combination $\mathsf{M}_{\mathcal{H}}$ of the real machines $\mathsf{M}_u$ from the combination $\mathsf{THSim}_{\mathcal{H}}$ of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ (for all sets $\mathcal{H}$ indicating the correct machines). We do not repeat the precise definition of "combinations" here.

The proof is essentially a bisimulation. This means to define a mapping between the states of two systems and a sufficient set of invariants so that one can show that every external input to the two systems (in mapped states fulfilling the invariants) keeps the system in mapped states fulfilling the invariants, and that outputs are identical. However, the states of both systems 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, etc. We circumvent this problem by conducting the proof via a combined system $\mathsf{C}_{\mathcal{H}}$, from which both original systems $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$ can be derived. The two derivations are two mappings, and we perform the two bisimulations in parallel. By the transitivity of indistinguishability (of the families of view of the same A and H in all three configurations), we obtain the desired result. This is shown in Figure 3.

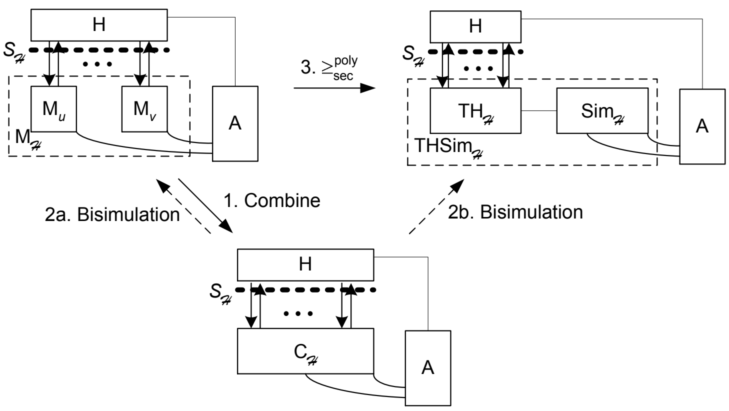


Figure 3: Overview of the Simulatability Proof.

Further, our bisimulation is probabilistic, as we will see in the invariant "strongly correct arguments". Moreover, certain "error sets" of runs remain where the bisimulation fails. We show that these sets have negligible probability (in the security parameter) at the end; this is sufficient for computational indistinguishability.

In addition to standard invariants, we have an information-flow invariant "word secrecy" which helps us to show that the adversary cannot guess certain values in these final proofs for the error sets. Although we can easily show that the probability of a truly random guess hitting an already existing value is negligible, we can only exploit this if *no* information (in the Shannon sense) about the particular value has been given to the adversary. We hence have to show that the adversary did not even receive any *partial* information about this value, which could be derivable since, e.g., the value was hidden within a nested term. Dealing with aspects of this kind is solved by incorporating static information-flow analysis in the bisimulation proof.

## 6.1 Combined System

The combined system mainly contains a database $D^*$ structured like $D$ in $\mathsf{TH}_\mathcal{H}$. An entry $x$ may have the following additional attributes:

- $x.word \in \{0,1\}^*$ contains real data as in $\mathsf{M}_\mathcal{H}$ or $\mathsf{Sim}_\mathcal{H}$ under the same handle(s).
- $x.parsed_u \in \{\mathsf{true}, \mathsf{false}\}$ for $u \in \mathcal{H}$ is $\downarrow$ if $x.hnd_u = \downarrow$; otherwise $\mathsf{true}$ indicates that the entry would be parsed in $D_u$, and $\mathsf{false}$ that it would still be of type $\mathsf{null}$. As entries of type $\mathsf{pka}$ do not exist in the real system, we always have $parsed_u = \downarrow$ for them.
- $x.owner$ for secret keys and authenticators is $\mathsf{adv}$ if the key or the authenticator was first received from the adversary, otherwise $\mathsf{honest}$.

Its state also contains variables $size$ and $curhnd_u$ as in $\mathsf{TH}_\mathcal{H}$, and all variables $steps_{\mathsf{p}?}$ as in $\mathsf{THSim}_\mathcal{H}$ are equal to the step counters in $\mathsf{M}_\mathcal{H}$. In the transitions of $\mathsf{C}_\mathcal{H}$, the $D$-part of the database $D^*$ and the variables $size$ and $curhnd_u$ are treated as in $\mathsf{TH}_\mathcal{H}$. An entry whose first handle $x.hnd_u$ is for $u \in \mathcal{H}$ gets the word that $\mathsf{M}_u$ would contain under this handle, and otherwise that from $\mathsf{Sim}_\mathcal{H}$. Thus, essentially, entries created due to basic commands from $\mathsf{H}$ get the words that $\mathsf{M}_u$ would construct, while words received in network inputs from $\mathsf{A}$ are parsed completely and entered as by $\mathsf{Sim}_\mathcal{H}$. Outputs to $\mathsf{H}$ are made as in $\mathsf{TH}_\mathcal{H}$, outputs to $\mathsf{A}$ as in $\mathsf{M}_\mathcal{H}$.

## 6.2 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 $\omega$ abbreviate word lookup, i.e., $\omega(i) := D^*[i].word$ if $i \in \mathcal{HNDS}$, else $\omega(i) := i$. Let $\omega^*$, applied to a list, denote that $\omega$ 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.

For a given state of $\mathsf{C}_\mathcal{H}$, we define derived states corresponding to the original systems. In the following, we only define the derivations for entries of our new types, and of those that occur in the upcoming proof.

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

$curhnd_u$ (for $u \in \mathcal{H} \cup \{\mathsf{a}\}$) and $size$: 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 $x^{\mathsf{hnd}} \leq curhnd_u$, 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 $type \in \{\mathsf{ska}, \mathsf{aut}\}$, then $D_u^* :\Leftarrow (x^{\mathsf{hnd}}, m, type, ())$.

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

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

- If $type = \mathsf{pka}$, let $ska^{\mathsf{ind}} := x + 1$. If $D^*[ska^{\mathsf{ind}}].owner = \mathsf{adv}$, then $D_\mathsf{a}^* :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{adv}))$, else $D_\mathsf{a}^* :\Leftarrow (x^{\mathsf{hnd}}, m, (\mathsf{honest}, \omega(ska^{\mathsf{ind}})))$.
- If $type \in \{\mathsf{ska}, \mathsf{aut}\}$, then $D_\mathsf{a}^* :\Leftarrow (x^{\mathsf{hnd}}, m, ())$.

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

### 6.3 Invariants in $\mathsf{C}_{\mathcal{H}}$

For the bisimulation, we need invariants about $\mathsf{C}_{\mathcal{H}}$. In the original library, there are invariants *index and handle uniqueness*, *well-defined terms*, *message correctness*, *key secrecy*, *no unparsed secret keys*, *length bounds*, *fully defined*, and *correct key pairs*. They are not explicitly used in our upcoming proof and it is easy to see that they remain correct for our extension to the library. In the following, we present the important invariants for the new proof. Each of them already occurred in [4], except for "correct verification", which trivially holds in the original proof since it only makes statements about our new types. Each existing invariant is generalized for dealing with our new types, leaving entries of old types unaffected.

- *Word uniqueness.* For each word $m \in \{0,1\}^*$, we have $|D^*[word = m \ \wedge \ type \notin \{\mathsf{sks}, \mathsf{ske}, \mathsf{pka}\}]| \leq 1$.

- *Correct length.* For all $i \leq size$, $D^*[i].len = |D^*[i].word|$, except if $D^*[i].type \in \{\mathsf{sks}, \mathsf{ske}, \mathsf{pka}\}$.

- *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 about which $\mathsf{A}$ may have some information. We claim that at all times, no information from outside has flowed into $Pub\_Var$ in the sense of information flow in static program analysis. The set $Pub\_Var$ contains
  - all words $D^*[i].word$ with $D^*[i].hnd_\mathsf{a} \neq \downarrow$;
  - the state of $\mathsf{A}$ and $\mathsf{H}$, and the $\mathsf{TH}_{\mathcal{H}}$-part of the state of $\mathsf{C}_{\mathcal{H}}$;
  - secret keys of public-key encryption and digital signature schemes 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 $D^*[i+1].word$.[3]

"Word secrecy" implies that no information from random values $sr$ in authentication keys or $r$ in authenticators has flowed into $Pub\_Var$ unless the respective entries have adversary handles. Absence of information flow in the static sense implies absence of Shannon information.

The remaining two invariants "correct arguments" and "strongly correct arguments" establish a relationship between the real message of an entry and its abstract type and arguments. For each type, there is a separate relationship. In the following, we introduce such a relationship for our new types.

[3]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 this does not apply for our key identifiers, as they carry no information about the corresponding secret key.

- *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 $arg^{\mathsf{real}} := \omega^*(arg^{\mathsf{ind}})$. If $type^{\mathsf{id}} \notin \{\mathsf{sks}, \mathsf{ske}, \mathsf{pka}\}$, let $(type, arg^{\mathsf{parse}}) := \mathsf{parse}(m)$, and we require $type = type^{\mathsf{id}}$, and:
  - If $type = \mathsf{aut}$, then $arg^{\mathsf{parse}} = arg^{\mathsf{real}}[1]$. (Parsing does not output the key identifiers.)
- *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:[4]
  - If $type = \mathsf{aut}$, then $arg^{\mathsf{ind}}$ is of the form $(l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}})$. Let $ska^{\mathsf{ind}} := pka_1^{\mathsf{ind}} + 1$ and $arg'^{\mathsf{real}} := \omega^*(ska^{\mathsf{ind}}, l^{\mathsf{ind}})$. Then $m \leftarrow \mathsf{make\_auth}(arg'^{\mathsf{real}})$.
  - If $type = \mathsf{ska}$, then $m \leftarrow \mathsf{make\_auth\_key}()$.

The following invariant is new, and deals with consistent verification in the ideal and real system.

- *Correct Verification.* For all $i, j \leq size$ with $D^*[i].type = \mathsf{aut}$ and $D^*[j].type = \mathsf{ska}$: Let $(\mathsf{aut}, sr, r, l, aut) := D^*[i].word$, $(l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}})) := D^*[i].arg$, and $(\mathsf{ska}, sk, sr') := D^*[j].word$. Then $pka^{\mathsf{ind}} := D^*[j].ind - 1 \in \{pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}}\}$ if and only if $sr = sr'$ and $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$.

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

**Definition 6.1** *(Bisimulation Property) By "an input retains all invariants" we mean the following:*

- *The resulting 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".*

$\diamond$

All conditions are obviously true initially when all databases are empty and the counters 0. The bisimulation shows that each input maintains the bisimulation property. Unfortunately, this is not true for *all* runs of the system, e.g., if two nonces collide in the generation of two different secret keys, or if the adversary successfully forges an authenticator. These runs are collected in *error sets.*

## 6.4 The Bisimulation

### 6.4.1 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

[4]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.

so is $\mathsf{C}_{\mathcal{H}}$'s output at $\mathsf{out}_u!$. We will not always mention this again. Moreover, "word secrecy" is clear since the output at $\mathsf{out}_u!$ and the updates to the $D$-part of $D^*$ are made entirely with commands from $\mathsf{TH}_{\mathcal{H}}$ and thus within $Pub\_Var$. New or existing words only get a handle for $u$, so that nothing is added to $Pub\_Var$.

- *Key generation:* $ska^{\mathsf{hnd}} \leftarrow \mathsf{gen\_auth\_key}()$.

  Both $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{M}_u$ set $ska^{\mathsf{hnd}} := curhnd_u\text{++}$, and make two entries in case of $\mathsf{TH}_{\mathcal{H}}$, respectively one entry in case of $\mathsf{M}_u$. In $\mathsf{C}_{\mathcal{H}}$ this gives $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{pka}, arg := (), len := 0)$ and $D^* :\Leftarrow (ind := size\text{++}, type := \mathsf{ska}, arg := (ind - 1), hnd_u := ska^{\mathsf{hnd}}, len := \mathsf{ska\_len}^*(k), parsed_u := \mathsf{true}, word := sk^*)$ where $sk^* \leftarrow \mathsf{make\_auth\_key}()$.

  The outputs are equal, and "correct derivation" is clear. If "word uniqueness" is not fulfilled, $sk^*$ matches an already existing value. In particular, the nonce $sr$ within $sk^*$ then equals an old one at the same place within a word, hence we put the run in an error set $Nonce\_Coll$.

  "Correct length" is fulfilled because $\mathsf{ska\_len}^*(k) = \mathsf{list\_len}(|\mathsf{ska}|, \mathsf{ska\_len}(k), \mathsf{nonce\_len}(k)) = |sk^*|$ by definition of $\mathsf{make\_auth\_key}$; nothing is required for type $\mathsf{pka}$. Under "correct arguments", nothing is required for type $\mathsf{ska}$ and $\mathsf{pka}$. "Strongly correct arguments" is obvious.

  If "correct verification" is not fulfilled, the new secret key is a valid authentication key for an existing authenticator. This in particular means that the newly generated nonce $sr$ in the new key equals an existing nonce in the authenticator. Hence, we put the run in an error set $Nonce\_Coll$.

- *Authenticator generation:* $aut^{\mathsf{hnd}} \leftarrow \mathsf{auth}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

  Let $ska^{\mathsf{ind}} := D^*[hnd_u = ska^{\mathsf{hnd}}].ind$ and $l^{\mathsf{ind}} := D^*[hnd_u = l^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{M}_u$ return $\downarrow$ if $D^*[ska^{\mathsf{ind}}].type \neq \mathsf{ska}$ or $D^*[l^{\mathsf{ind}}].type \neq \mathsf{list}$. Their tests are equivalent by "correct derivation".

  Further, $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if $length := \mathsf{aut\_len}^*(k, D^*[l^{\mathsf{ind}}].len) > \mathsf{max\_len}(k)$. Else it sets $aut^{\mathsf{hnd}} := curhnd_u\text{++}$ and makes a new entry $D :\Leftarrow (ind := size\text{++}, type := \mathsf{aut}, arg := (l^{\mathsf{ind}}, ska^{\mathsf{ind}} - 1), hnd_u := aut^{\mathsf{hnd}}, len := length)$.

  $\mathsf{M}_u$ uses $(sk^*, l) := \omega(sk^{\mathsf{ind}}, l^{\mathsf{ind}})$ and sets $aut^* \leftarrow \mathsf{make\_aut}(sk^*, l)$. If $|aut^*| > \mathsf{max\_len}(k)$, it returns $\downarrow$. This length test equals that in $\mathsf{TH}_{\mathcal{H}}$: By "strongly correct arguments", the key $sk^*$ was generated with $\mathsf{make\_auth\_key}()$. With the notation from inside $\mathsf{make\_auth}$, this means that $sk$ was correctly generated, and thus we have $|aut| = \mathsf{aut\_len}(k, |(r, l)|) = \mathsf{aut\_len}'(k, |l|)$. This yields $|aut^*| = \mathsf{aut\_len}^*(k, |l|)$, and by "correct length" for the entry $D^*[l^{\mathsf{ind}}]$ 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 $\mathsf{M}_u$ sets $aut^{\mathsf{hnd}} := curhnd_u\text{++}$ and makes an entry $D_u :\Leftarrow (aut^{\mathsf{hnd}}, aut^*, \mathsf{aut}, ())$.

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

  Now we consider the new authenticator entry: "Correct derivation" is clear if we augment $\mathsf{TH}_{\mathcal{H}}$'s entry with the word $aut^*$ and $parsed_u = \mathsf{true}$. If "word uniqueness" is not fulfilled, then $r$ within $aut^*$ equals an old value in the same place in a word; hence we put the run in the error set $Nonce\_Coll$. "Correct length" is fulfilled as shown above. "Correct arguments" follows by comparing the output format of $\mathsf{make\_auth}$ with the predicate in

$\mathsf{parse\_aut}$. "Strongly correct arguments" holds by construction. If "correct verification" is not fulfilled, then we again have a nonce collision as the nonce within the new authenticator matches an existing one within a key. Hence, the run is put into the error set $Nonce\_Coll$.

- *Authenticator verification:* $v \leftarrow \mathsf{auth\_test}(aut^{\mathsf{hnd}}, ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$. Let $aut^{\mathsf{ind}} := D^*[hnd_u = aut^{\mathsf{hnd}}].ind$ and $ska^{\mathsf{ind}} := D^*[hnd_u = ska^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{M}_u$ return $\downarrow$ if $D^*[aut^{\mathsf{ind}}].type \neq \mathsf{aut}$ or if $D^*[ska^{\mathsf{ind}}].type \neq \mathsf{ska}$ (indeed $\mathsf{M}_u$ has parsed the entries). Otherwise, let $(l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}}) := D^*[aut^{\mathsf{ind}}].arg$, $(\mathsf{aut}, sr, r, l, aut) := D^*[aut^{\mathsf{ind}}].word$, and $(\mathsf{ska}, sk, sr) := D^*[ska^{\mathsf{ind}}].word$. By "correct arguments" for the entry $D^*[aut^{\mathsf{ind}}]$, we have $l = \omega^*(l^{\mathsf{ind}})$, and hence $l^{\mathsf{hnd}} = D^*[l^{\mathsf{ind}}].hnd_u$ if and only if $l \neq D_u[l^{\mathsf{hnd}}].word$. $\mathsf{TH}_{\mathcal{H}}$ outputs $\mathsf{false}$ if $pka_1^{\mathsf{ind}} = \downarrow$ or $ska^{\mathsf{ind}} - 1 \notin \{pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}}\}$, and $\mathsf{true}$ otherwise. $\mathsf{M}_u$ outputs $\mathsf{false}$ iff $sr \neq D_u[ska^{\mathsf{hnd}}].word[3]$ or $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{false}$. This is equivalent by "correct verification" and "correct derivation". No invariants are affected here.

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

  We start exactly as in authenticator verification: Let $aut^{\mathsf{ind}} := D^*[hnd_u = aut^{\mathsf{hnd}}].ind$. Both $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{M}_u$ return $\downarrow$ if $D^*[aut^{\mathsf{ind}}].type \neq \mathsf{aut}$. (Indeed $\mathsf{M}_u$ has parsed the entry.) Otherwise, let $(l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}}) := D^*[aut^{\mathsf{ind}}].arg$, $aut^* := D^*[aut^{\mathsf{ind}}].word$, and $(l) \leftarrow \mathsf{parse\_aut}(aut^*)$. By "correct arguments" for the entry $D^*[aut^{\mathsf{ind}}]$, we have $l = \omega(l^{\mathsf{ind}})$.

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

### 6.4.2 Send Command from Honest User

We now consider an input $\mathsf{send\_i}(v, l_u^{\mathsf{hnd}})$ at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$ (the list $l_u^{\mathsf{hnd}}$ should be sent to $v$). 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 $\mathsf{TH}_{\mathcal{H}}$.

Let $l^{\mathsf{ind}} := D^*[hnd_u = l_u^{\mathsf{hnd}}].ind$. Now $\mathsf{M}_u$ always outputs $l := D^*[l^{\mathsf{ind}}].word$. An inductive proof is used that $\mathsf{id2real}$ retains all invariants and produces the right outputs. By inspection of $\mathsf{id2real}$, we see that the first three steps of the algorithm are essentially independent of the type of the considered entry (up to domain checks which are fulfilled by construction when interacting with $\mathsf{TH}_{\mathcal{H}}$). In step 4, $\mathsf{id2real}$ then 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 $add\_arg$.

In [4], it has been proven (in Lemma 7.6) that it is sufficient to show:

- a correct result $m = m^*$, where $m^*$ is the word the $\mathsf{M}_u$ produces, i.e., $m^* := D^*[m^{\mathsf{ind}}].word$. We further can assume "strongly correct arguments" for $m^*$.

- "correct derivation" of $add\_arg$ in the new entry;

- "word secrecy" for $m$, i.e., no flow of secret information into $m$, where arguments $m_i$ are not secret information.

For our new types, these conditions are also sufficient, which can be proven analogously to the original proof. Since the proof mainly relies on a thorough investigation of the first three steps of id2real, we have to omit the details here due to lack of space.

**Authentication Keys** If $type = \mathsf{pka}$, then id2real sets $sk^* \leftarrow \mathsf{make\_auth\_key}()$, $m := \epsilon$ and $add\_arg := (\mathsf{honest}, sk^*)$.

Let $m^{\mathsf{ind}} := D^*[m^{\mathsf{hnd}}].ind$ and $ska^{\mathsf{ind}} = m^{\mathsf{ind}} + 1$ and $sk^{*\mathsf{real}} := D^*[ska^{\mathsf{ind}}].word$. By "strongly correct arguments" $sk^{*\mathsf{real}}$ was chosen with $\mathsf{make\_auth\_key}()$. Moreover, we have $\mathsf{a} \notin \mathsf{owners}(D^*[ska^{\mathsf{ind}}])$, because otherwise $D^*[m^{\mathsf{ind}}]$ would also have got an a-handle at once. In the derived $D^*_{\mathsf{a}}$, we therefore have an entry $(m^{\mathsf{hnd}}, m, (\mathsf{honest}, sk^{*\mathsf{real}}))$ with the same distribution as id2real's choice. "Word secrecy" is clear since $m = \epsilon$.

For $type = \mathsf{ska}$, Let $pka^{\mathsf{hnd}} := m_1^{\mathsf{hnd}}$. By construction, we have $D^*[pka^{\mathsf{hnd}}].type = \mathsf{pka}$.[5] Let $pk^{\mathsf{ind}} := D^*[pka^{\mathsf{hnd}}].ind$, $ska^{\mathsf{ind}} = pka^{\mathsf{ind}} + 1$. Analogously to the type pka, we know that $\mathsf{a} \notin \mathsf{owners}(D^*[ska^{\mathsf{ind}}])$, and with "correct derivation" we obtain $D^*_{\mathsf{a}}[pka^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, \omega(ska^{\mathsf{ind}}))$. Now the output is $m := \omega(sk^{\mathsf{ind}})$, which is equal to the output $D^*[sk^{\mathsf{ind}}].word$ in the real system. "Word secrecy" is clear.

**Authenticators** If $type = \mathsf{aut}$, "strongly correct arguments" implies that $arg^{\mathsf{ind}}$ is of the form $(l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}})$ with $pka_1^{\mathsf{ind}} \neq \downarrow$. This proves the format claim in id2real.

Let $ska^{\mathsf{ind}} := pka_1{}^{\mathsf{ind}} + 1$ and $(sk^*, l^*) := \omega(ska^{\mathsf{ind}}, l^{\mathsf{ind}})$. By "strongly correct arguments", $m^*$ is distributed as $m^* \leftarrow \mathsf{make\_auth}(sk^*, l^*)$. If $D_{\mathsf{a}}[pka_1^{\mathsf{ind}}].add\_arg[1] = \mathsf{honest}$, then "correct derivation" of $D_{\mathsf{a}}$ implies $D_{\mathsf{a}}[pka_1^{\mathsf{hnd}}].add\_arg = (\mathsf{honest}, sk^*)$, where $pka_1^{\mathsf{hnd}} = D^*[pka_1{}^{\mathsf{ind}}]$. If $D_{\mathsf{a}}[pka_1^{\mathsf{ind}}].add\_arg = (\mathsf{adv})$ then "correct derivation" of $D_{\mathsf{a}}$ implies $D_{\mathsf{a}}[pka_1^{\mathsf{ind}} + 1].word = sk^*$. In both cases, id2real sets $m \leftarrow \mathsf{make\_auth}(sk^*, l^*)$. This is the same distribution.

For proving "word secrecy" for $m$, we only have to consider the parameter $sk^*$, because $l^*$ is a parameter $m_1$ (and make_auth is functional). By definition of "word secrecy", $sk^*$ already belongs to $Pub\_Var$, hence "word secrecy" is clear.

### 6.4.3 Network Input from the Adversary

We now consider the effects of an input $l$ from A. Recall that on such an input $\mathsf{C}_{\mathcal{H}}$ acts entirely like $\mathsf{THSim}_{\mathcal{H}}$. Both $\mathsf{M}_u$ and $\mathsf{Sim}_{\mathcal{H}}$ 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. Using a lemma from [4], we only have to show 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 $D^*[hnd_{\mathsf{a}} = l_{\mathsf{a}}^{\mathsf{hnd}}].type \notin \{\mathsf{sks}, \mathsf{ske}\}$.
- "Correct derivation" of $D_{\mathsf{a}}$ 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.

The lemma carries over to our new types with marginal extensions of the proof.

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

[5] This could as well be treated as an invariant, but it is obvious since secret keys always have their key identifier as only argument by definition, and their argument never changes.

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

Finally it sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add\_arg)$.

We therefore have to show the postulated properties for our new type-specific algorithms together with those last two assignments.

**Authentication Keys** The algorithm $\mathsf{real2id\_ska}(m, ())$ calls $ska^{\mathsf{hnd}} \leftarrow \mathsf{gen\_auth\_key}()$ at $\mathsf{in_a}!$ and sets $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}$++$, \epsilon, (\mathsf{adv}))$ for the key identifier and $add\_arg := ()$ for the secret key.

Recall that the upcoming loop over all $aut^{\mathsf{hnd}}$ can only modify the database $D^*$ by outputting a command $\mathsf{adv\_fix\_aut\_validity}$, which does not create new entries.

Hence $\mathsf{TH}_{\mathcal{H}}$ also makes only two new entries with $pka^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ and $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++.

In $\mathsf{C}_{\mathcal{H}}$, the key identifier entry results in $D^* :\Leftarrow (ind := size$++$, type := \mathsf{pka}, arg := (), hnd_{\mathsf{a}} := pka^{\mathsf{hnd}}, len := 0)$. The secret-key entry results in $D^* :\Leftarrow (ind := size$++$, type := \mathsf{ska}, arg := (ind - 1), hnd_{\mathsf{a}} := m^{\mathsf{hnd}}, len := \mathsf{ska\_len}^*(k), word := m)$. It fulfills the postulated output conditions. Here "correct derivation", "correct length", "word secrecy" and "correct arguments" are clear. If "Word uniqueness" is not fulfilled, then there exists a prior entry $x \in D^*$ with $x.word = m$, i.e., the adversary has guessed a key which it has not seen yet. This especially implies that he has guessed the nonce $sr$, hence we put this run in an error set $Nonce\_Guess$. We have $x.hnd_{\mathsf{a}} = \downarrow$ by "correct derivation" of $D_{\mathsf{a}}$, because $m$ is not present in $D_{\mathsf{a}}$. Thus, $x.word \notin Pub\_Var$.

Let $pk^{\mathsf{ind}} := size - 1$. Then "correct derivation" holds because $sk^{\mathsf{ind}} := pk^{\mathsf{ind}} + 1$ designates the secret-key entry with $D^*[sk^{\mathsf{ind}}].owner = \mathsf{adv}$, so that $add\_arg = (\mathsf{adv})$ is the correct choice in $D_{\mathsf{a}}$. "Correct arguments" and "word secrecy" are obvious. For "correct length", nothing is required for type $\mathsf{pka}$. "Word uniqueness" need not be shown for this entry.

We now consider the for-loop, which checks if already existing authenticators are valid for the new key. Let $sk^* := m = (\mathsf{ska}, sk, sr)$, and assume that there exists a handle $aut^{\mathsf{hnd}}$ with $D_{\mathsf{a}}[aut^{\mathsf{hnd}}].type = \mathsf{aut}$ and $D_{\mathsf{a}}[aut^{\mathsf{hnd}}].word = (\mathsf{aut}, sr, r, l, aut)$ for $r \in \{0,1\}^{\mathsf{nonce\_len}(k)}$, $l \in \{0,1\}^+$, $aut \in \{0,1\}^{\mathsf{aut\_len}'(k,|l|)}$, and $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$. Then $\mathsf{Sim}_{\mathcal{H}}$ calls $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, aut^{\mathsf{hnd}})$. Now $\mathsf{TH}_{\mathcal{H}}$ returns $\downarrow$ if $aut := D[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}} \wedge type = \mathsf{aut}].ind = \downarrow$ or if $ska := D[hnd_u = ska^{\mathsf{hnd}} \wedge type = \mathsf{ska}].ind = \downarrow$. "Correct derivation" for the authenticator entry and parsing of the secret key imply that these checks succeed.

Now let $(l, pka_1, \ldots, pka_j) := D[aut].arg$ and $pka := ska - 1$. If $pka \notin \{pka_1, \ldots, pka_j\}$ set $D[aut].arg =: (l, pka_1, \ldots, pka_j, pka)$. Here "correct derivation", "correct length", "word uniqueness", and "word secrecy" are clear. "Correct arguments" is also clear due to the special format of type $\mathsf{aut}$ (parsing does not output the key identifiers). If $pka \neq \downarrow$ "strongly correct arguments" is unaffected. Otherwise the authenticator has been created by a command $\mathsf{adv\_unknown\_auth}$, hence $\mathsf{a} \in \mathsf{owners}(D^*[pka_1 + 1])$.

The only invariant left to show is "correct verification". Let $i \leq size$ with $D^*[i].type = \mathsf{aut}$ and $D^*[i].word = (\mathsf{aut}, sr, r, l, aut)$ that fits to the key $sk^*$, i.e, $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$. Let $aut^{\mathsf{hnd}} := D^*[i].hnd_{\mathsf{a}}$. Because of the checks of $\mathsf{Sim}_{\mathcal{H}}$, it is sufficient to show that the corresponding key identifier is added to the authenticator's arguments. We distinguish two cases: If $\mathsf{a} \in \mathsf{owners}(D^*[i])$, then this entry is present in $D_{\mathsf{a}}$, hence $\mathsf{Sim}_{\mathcal{H}}$ outputs $\mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, aut^{\mathsf{hnd}})$. The checks of $\mathsf{TH}_{\mathcal{H}}$ will succeed since they correspond to the checks of $\mathsf{Sim}_{\mathcal{H}}$ by "correct derivation". Hence if $ska^{\mathsf{hnd}} - 1$ is not contained in the element list, it is added, which retains the invariant. If $\mathsf{a} \notin \mathsf{owners}(D^*[i])$, i.e., the key fits to an authenticator that the adversary has not seen yet, he especially has not seen the nonce $r$. Hence, we put this run in an error set *Nonce_Guess*. Because of $\mathsf{a} \notin \mathsf{owners}(D^*[i])$, we have $aut^{\mathsf{hnd}} = \downarrow$, hence $D^*[aut^{\mathsf{hnd}}].word \notin Pub\_Var$.

**Authenticators** When $\mathsf{real2id\_aut}(m, (l))$ is called, we know from parsing that $l$ is a tagged list and shorter than $m$, so that also $|l| \leq \mathsf{max\_len}(k)$. Moreover, $l \in Pub\_Var$ because they were generated from $m \in Pub\_Var$ by the functional algorithm $\mathsf{parse}$. Hence when $\mathsf{real2id\_aut}$ starts with a recursive call $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$; this call fulfill the postulated conditions by induction hypothesis. Thus, it retains all invariants and ensures $D^*[hnd_{\mathsf{a}} = l^{\mathsf{hnd}}].word = l$. Let $l^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = l^{\mathsf{hnd}}].ind$ and $m = (\mathsf{aut}, sr, r, l, aut)$. Let $Ska := \{ska^{\mathsf{hnd}} \mid D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].type = \mathsf{ska} \wedge D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].word[3] = sr \wedge \mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$ for $sk := D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].word[2]\}$.

**Case 1: Transformed Authenticator.** $\mathsf{Sim}_{\mathcal{H}}$ first verifies whether the adversary has already seen another authenticator from an honest user for the same message. It sets $Aut := \{aut^{\mathsf{hnd}} \mid D^*[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}}].word = (\mathsf{aut}, sr, r', l, aut') \wedge D^*[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}}].type = \mathsf{aut}\}$. For each $aut^{\mathsf{hnd}} \in Aut$, it sets $(\mathsf{aut}, arg_{aut^{\mathsf{hnd}}}) \leftarrow \mathsf{adv\_parse}(aut^{\mathsf{hnd}})$ and $pka_{aut^{\mathsf{hnd}}} := arg_{aut^{\mathsf{hnd}}}[2]$.

Now assume for contradiction that there exist two such distinct elements $pka^{\mathsf{hnd}}_{aut_1^{\mathsf{hnd}}}, pka^{\mathsf{hnd}}_{aut_2^{\mathsf{hnd}}}$ with $D^*[hnd_{\mathsf{a}} = pka^{\mathsf{hnd}}_{aut_1^{\mathsf{hnd}}}].add\_arg[1] = D^*[hnd_{\mathsf{a}} = pka^{\mathsf{hnd}}_{aut_2^{\mathsf{hnd}}}].add\_arg[1] = \mathsf{honest}$. Let $pka_1 := D^*[hnd_{\mathsf{a}} = pka^{\mathsf{hnd}}_{aut_1^{\mathsf{hnd}}}].ind$ and $pka_2 := D^*[hnd_{\mathsf{a}} = pka^{\mathsf{hnd}}_{aut_2^{\mathsf{hnd}}}].ind$. Since $D^*[hnd_{\mathsf{a}} = aut_1^{\mathsf{hnd}}].variword[2] = D^*[hnd_{\mathsf{a}} = aut_2^{\mathsf{hnd}}].word[2] = sr$, we have $D_{\mathsf{a}}[pka_1 + 1].word[2] = D_{\mathsf{a}}[pka_2 + 1].word[2] = sr$. But "correct derivation" implies that $D_{\mathsf{a}}[pka_1 + 1].owner = D_{\mathsf{a}}[pka_2 + 1].owner = \mathsf{honest}$, so "strongly correct arguments" implies that $D_{\mathsf{a}}[pka_1 + 1].word$ and $D_{\mathsf{a}}[pka_2 + 1].word$ have been created by the command $\mathsf{make\_auth\_key}$. This means that if two such distinct elements existed, the nonces $sr$ collided in two executions of $\mathsf{make\_auth\_key}$. In this case, we put the run into an error set *Nonce_Coll*.

Now assume that there exists a unique $pka_{aut^{\mathsf{hnd}}}$, and let $sk^* = D_{\mathsf{a}}[pka_{aut^{\mathsf{hnd}}}].add\_arg[2]$. Then the simulator checks if $\mathsf{atest}_{sk^*[2]}(aut, (r, l)) = \mathsf{true}$, i.e., it only continues the interaction with $\mathsf{TH}_{\mathcal{H}}$ if the check in the real system is correct. This is equivalent by "correct verification". In this case, it calls $trans\_aut^{\mathsf{hnd}} \leftarrow \mathsf{adv\_transform\_aut}(aut^{\mathsf{hnd}})$ at $\mathsf{in_a}!$ and sets $add\_arg := ()$.

Let $aut^{\mathsf{ind}} := D^*[hnd_{\mathsf{a}} = aut^{\mathsf{hnd}}].ind$. By "correct derivation", we have $D^*[aut^{\mathsf{ind}}].type = \mathsf{aut}$. With the preconditions about $aut^{*\mathsf{real}}$, "correct arguments" for $aut^{\mathsf{ind}}$, and "word uniqueness" for $l$, this implies $D^*[aut^{\mathsf{ind}}].arg = (l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}})$. Hence $\mathsf{TH}_{\mathcal{H}}$ sets $trans\_aut^{\mathsf{hnd}} := curhnd_{\mathsf{a}}{+}{+}$ and makes a new entry. Together with the new entry in $D_{\mathsf{a}}$, this results in $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{aut}, arg := (l^{\mathsf{ind}}, pka_1^{\mathsf{ind}}), hnd_{\mathsf{a}} := aut^{\mathsf{hnd}}, len := D^*[aut^{\mathsf{ind}}].len, word := m)$. "Correct derivation" is clearly retained, and the postulated output condition is fulfilled. "Correct arguments" holds because we showed that the arguments copied from $D^*[aut^{\mathsf{ind}}]$ are those that we get by parsing $m$. For "correct length", we use that $D^*[aut^{\mathsf{ind}}].len = |aut^{*\mathsf{real}}|$ by "correct length" for $aut^{\mathsf{ind}}$. Thus we only have to show $|m| = |aut^{*\mathsf{real}}|$. This holds because both parse as authenticators with the same component $l$. "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 $D_{\mathsf{a}}$. This means that the adversary has guessed an authenticator that existed in $\mathsf{TH}_{\mathcal{H}}$ but that he has not been send yet. This in particular means that he has guessed the inherent nonce $r$ within $m$, hence we put the run in the error set *Nonce_Guess*. We have

$x.hnd_{\mathsf{a}} = \downarrow$, hence $x \notin \mathit{Pub\_Var}$.

After that, $\mathsf{Sim}_{\mathcal{H}}$ calls $\mathsf{adv\_fix\_aut\_validity}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ for every $ska^{\mathsf{hnd}} \in \mathit{Ska}$, i.e., it enters the key identifiers for the valid secret keys. The only invariant that could be affected is "correct verification". We distinguish two cases: First, we assume that if an entry $i$ in $D^*$ exists with $D^*[i].type = \mathsf{ska}$, $sk^* := (\mathsf{ska}, sk, sr) := D^*[i].word$, and $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$, then there is an entry $j$ with $j.hnd_{\mathsf{a}} \neq \downarrow$ that also fulfills these conditions. In this case, a handle $ska^{\mathsf{hnd}}$ for $j$ will be contained in $\mathit{Ska}$ by "correct derivation" and hence $\mathsf{Sim}_{\mathcal{H}}$ calls $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(aut^{\mathsf{hnd}}, ska^{\mathsf{hnd}})$. Then "correct verification" follows analogously to the proof of the previous subsection for authentication keys. Secondly, if there exists an entry $i$ in $D^*$ that meets the above requirements, but for all entries $j$ of the above form we have $j.hnd_{\mathsf{a}} = \downarrow$, then the adversary has guessed a valid authenticator, which means that in particular, he has guessed the inherent nonce $r$. We hence put the run into the error set *Nonce_Guess*. We again obtain $i \notin \mathit{Pub\_Var}$ because of $i.hnd_{\mathsf{a}} = \downarrow$.

**Case 2: A Valid Key Exists in $D_{\mathsf{a}}$.** We now consider the behavior of $\mathsf{Sim}_{\mathcal{H}}$ if $m$ is not a transformed authenticator, but $\mathsf{Sim}_{\mathcal{H}}$ finds a suitable secret key for testing the authenticator, i.e., we have $\mathit{Ska} \neq \emptyset$. $\mathsf{Sim}_{\mathcal{H}}$ then picks $ska^{\mathsf{hnd}} \in \mathit{Ska}$ arbitrarily, and calls $aut^{\mathsf{hnd}} \leftarrow \mathsf{auth}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.

$\mathsf{TH}_{\mathcal{H}}$ sets $ska := D^*[ska^{\mathsf{hnd}}].ind$, $l := D^*[l^{\mathsf{hnd}}].ind$ and outputs $\downarrow$ if $D^*[ska^{\mathsf{hnd}}].type \neq \mathsf{ska}$ or $D^*[l^{\mathsf{hnd}}].type \neq \mathsf{list}$. These checks are identical to the check of $\mathsf{Sim}_{\mathcal{H}}$ in case of $\mathsf{ska}$ and to parsing $m$ in case of $\mathsf{list}$, hence the checks succeed by "correct derivation". Now $\mathsf{TH}_{\mathcal{H}}$ sets $length := \mathsf{aut\_len}^*(k, D[l])$ and aborts if $length > \mathsf{max\_len}(k)$. This is equivalent to $\mathsf{Sim}_{\mathcal{H}}$'s checks since we know from parsing that $|m| = \mathsf{list\_len}(|\mathsf{aut}|, \mathsf{nonce\_len}(k), \mathsf{nonce\_len}(k), |l|, \mathsf{aut\_len}'(k, |l|)) = \mathsf{aut\_len}^*(k, |l|)$ and from "correct length" that $D^*[l^{\mathsf{ind}}].len = |l|$. Hence, $\mathsf{TH}_{\mathcal{H}}$ makes a new entry; in $\mathsf{C}_{\mathcal{H}}$ this yields $D :\Leftarrow (ind := size{+}{+}, type := \mathsf{aut}, arg := (l, ska - 1), hnd_{\mathsf{a}} := aut^{\mathsf{hnd}}, len := length, word = m)$. "Correct derivation", "Correct arguments" are clear; "correct length" holds as shown above. "Word secrecy" need not be shown for this entry. If "word uniqueness" is not fulfilled, then $m$ matches an existing authenticator entry $x$ in $D^*$. Similar to the previous case, we have $x.hnd_{\mathsf{a}} = \downarrow$ since $x$ does not exist in $D_{\mathsf{a}}$, hence

the nonce $r$ within $m$ must have been guessed. Hence we put the run into an error set *Nonce_Guess*. Because of $x.hnd_{\mathsf{a}} = \downarrow$ we have $x \notin \mathit{Pub\_Var}$.

After that, $\mathsf{Sim}_{\mathcal{H}}$ calls $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(ska'^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ for every $ska'^{\mathsf{hnd}} \in \mathit{Ska} \setminus \{ska^{\mathsf{hnd}}\}$, i.e., it enters the key identifiers for the valid secret keys. The only invariant that could be affected is "correct verification". We distinguish three cases: First, we assume that if an entry $i$ in $D^*$ exists with $D^*[i].type = \mathsf{ska}$, $sk^* := (\mathsf{ska}, sk, sr) := D^*[i].word$, and $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$, then there is an entry $j$ with $j.hnd_{\mathsf{a}} \neq \downarrow$ that also fulfills these conditions. In this case, a handle $ska'^{\mathsf{hnd}}$ for $j$ will be contained in $\mathit{Ska}$ by "correct derivation" and hence $\mathsf{Sim}_{\mathcal{H}}$ calls $v \leftarrow \mathsf{adv\_fix\_aut\_validity}(aut^{\mathsf{hnd}}, ska'^{\mathsf{hnd}})$. Then "correct verification" follows analogously to the proof of the previous subsection for authentication keys. Secondly, if there exists an entry $i$ in $D^*$ that meets the above requirements, but for all entries $j$ of the above form we have $j.hnd_{\mathsf{a}} = \downarrow$,

then the adversary has guessed a valid authenticator, which means that in particular, it has guessed the inherent nonce $r$. We hence put the run into the error set *Nonce_Guess*. We again obtain $i \notin \mathit{Pub\_Var}$ because of $i.hnd_{\mathsf{a}} = \downarrow$. Thirdly, no such entry $i$ exists in $D^*$. In the case, the adversary has produced a valid forgery for an (unknown) key of an honest user. Hence, we put the run in an error set *Auth_Forge*. We designate the forgery $(sk, aut, (r, l))$. Note that $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$ because this was verified when parsing $m$, and that $a \notin \mathsf{owners}(D^*[i])$.

Further, "strongly correct arguments" for $D^*[i]$ implies that $sk^*$ was chosen in $\mathsf{gen\_auth\_key}$, and thus as $sk \leftarrow \mathsf{gen_A}(1^k)$.

**Case 3: No Valid Key Exists in $D_\mathsf{a}$.** Now assume that $Ska = \emptyset$. This either means that no key in $D_\mathsf{a}$ has a suitable nonce $sr$ or that the authenticator test fails for all keys in $D_\mathsf{a}$. In all these cases, the command $\mathsf{adv\_unknown\_aut}(l^\mathsf{hnd})$ is used to create a new authenticator for $l$ within $\mathsf{TH}_\mathcal{H}$ but currently without any key identifier. $\mathsf{TH}_\mathcal{H}$ returns $\downarrow$ if $l := D[hnd_\mathsf{a} = l^\mathsf{hnd} \wedge type = \mathsf{list}].ind = \downarrow$ or $length := \mathsf{aut\_len}^*(k, D[l].len) > \mathsf{max\_len}(k)$. This is equivalent to $\mathsf{Sim}_\mathcal{H}$'s checks as shown in the previous case. $\mathsf{TH}_\mathcal{H}$ now creates a new entry, corresponding to the following entry in $\mathsf{C}_\mathcal{H}$: $D :\Leftarrow (ind := size{+}{+}, type := \mathsf{aut}, arg := (l), hnd_\mathsf{a} := aut^\mathsf{hnd}, len := length, word = m)$. "Correct derivation" and "Correct arguments" are clear; "correct length" holds as shown above. "Word secrecy" need not be shown for this entry. If "correct verification" is not fulfilled, we can show similarly to the above case, that this authenticator is valid for an existing key entry $x$ of an honest user, which is not yet present in the database $D_\mathsf{a}$. Hence, we put the run in the error set *Nonce_Coll* if $x$ is present in $D^*$ (i.e., does not have an adversary handle yet) and in*Auth_Forge* otherwise. Let again $sk^* := (\mathsf{ska}, sk, sr) := x.word$. We then designate the forgery $(sk, aut, (r, l))$, and we have $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$ because this was verified when parsing $m$, and that $a \notin \mathsf{owners}(D^*[x])$. Further, "strongly correct arguments" for $D^*[x]$ imply that $sk^*$ was chosen in $\mathsf{gen\_auth\_key}$, and thus as $sk \leftarrow \mathsf{gen_A}(1^k)$.

## 6.5 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 $(Auth\_Forge_k)_{k\in\mathbb{N}}$. 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*, *Nonce_Guess*, and *Auth_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 $\mathsf{H}$ and adversary $\mathsf{A}$.

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 $\mathsf{H}$ and $\mathsf{A}$ is polynomial-time. This has already been shown for the original library, and this also holds our extension, since each new transition is surely polynomial-time, and the number of interactions of $\mathsf{TH}_\mathcal{H}$ and $\mathsf{Sim}_\mathcal{H}$ in one transition is always polynomially bounded, cf. Section 5.4.

### 6.5.1 Nonce Collisions

The error set *Nonce_Coll* occurs in Sections 6.4.1 for the nonce-components $sr$ in authentication keys and $r$ in authenticators. A run is put into this set if a new nonce, created randomly as $sr \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce\_len}(k)}$ (similar for $r$), 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 is negligible. As there are only polynomially many such pairs, the overall probability is also negligible.

### 6.5.2 Nonce Guessing

The error set *Nonce_Guess* occurs in Section 6.4.3 and 6.4.3. 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 $x.hnd_{\mathsf{a}} = \downarrow$, $x.word \notin Pub\_Var$, and $x.type \in \{\mathsf{ska}, \mathsf{aut}\}$. "Strongly correct arguments" implies that each of them contains a nonce part generated as $sr \stackrel{\mathcal{R}}{\leftarrow} \{0,1\}^{\mathsf{nonce\_len}(k)}$ for type $\mathsf{ska}$ and $r \stackrel{\mathcal{R}}{\leftarrow} \{0,1\}^{\mathsf{nonce\_len}(k)}$ for type $\mathsf{aut}$. "Word secrecy" means that no information flowed from $sr$ (respectively $r$) into $Pub\_Var$, which is a superset of the information known to the adversary $\mathsf{A}$. 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.

### 6.5.3 Authenticator Forgery

The error set $Auth\_Forge$ occurs in Section 6.4.3 for authenticator forgeries. In the runs put into this set we designated a triple $(sk, (r,l), aut)$ with $\mathsf{atest}_{sk}(aut, (r,l)) = \mathsf{true}$ for a key $sk$ chosen as $sk \leftarrow \mathsf{gen_A}(1^k)$.

In the combined system $\mathsf{C}_{\mathcal{H}}$, this secret key $sk$ was a component $D^*[sk^{\mathsf{ind}}].word[2]$ with $a \notin \mathsf{owners}(D^*[sk^{\mathsf{ind}}])$. Thus it is only used if the command $\mathsf{auth}$ is entered at a port $\mathsf{in}_v?$ for $v \in \mathcal{H}$, and there within normal authentication $aut \leftarrow \mathsf{auth}_{sk}((r,l))$. Further, if $(r,l)$ had ever been signed with $sk$ before, the command $\mathsf{auth}$ would lead to an entry $x \in D^*$ with $x.type = \mathsf{aut}$ and $x.word$ of the form $(\mathsf{aut}, sr, r, l, aut')$. However, the existence of such an entry was excluded in the conditions for putting the run in the set $Auth\_Forge$. Thus we have indeed a valid forgery for the underlying authentication system.

This argument was almost a rigorous reduction proof already: We construct an adversary $\mathsf{A_{aut}}$ against the signer machine $\mathsf{Aut}$ from Definition 4.1 by letting $\mathsf{A_{Aut}}$ execute $\mathsf{C}_{\mathcal{H}}$, using the given $\mathsf{A}$ and $\mathsf{H}$ as blackboxes. It only has to choose an index $i \stackrel{\mathcal{R}}{\leftarrow} \{1, \ldots, n \cdot \mathsf{max\_in}(k)\}$ indicating for which of the up to $n \cdot \mathsf{max\_in}(k)$ authentication keys generated due to inputs at ports $\mathsf{in}_u?$ with $u \in \mathcal{H}$ it uses $sk$ obtained from the signer machine $\mathsf{Aut}$ instead. Hence the success probability of $\mathsf{A_{aut}}$ for each $k$ is at least $(n \cdot \mathsf{max\_in}(k))^{-1}$ (from guessing $i$ correctly) times the probability of $Auth\_Forge_k$. Hence the security of the authentication scheme implies that the probability of the sets $Auth\_Forge_k$ is negligible.

## References


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

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

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology: CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[6] R. Canetti. A unified framework for analyzing security of protocols. IACR Cryptology ePrint Archive 2000/067, Dec. 2001. `http://eprint.iacr.org/`.

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

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

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

[18] B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic security of reactive systems. Presented at the DERA/RHUL Workshop on Secure Architectures and Information Flow, 1999, Electronic Notes in Theoretical Computer Science (ENTCS), March 2000. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm`.

[19] 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.

[20] 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.

[21] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proc. 8th IEEE Computer Security Foundations Workshop (CSFW)*, pages 98–107, 1995.

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