Symmetric Authentication within a Simulatable Cryptographic Library

Michael Backes, Birgit Pfitzmann, and 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., [17,16,13,22,23,1,15,18].

The typical abstraction is the Dolev-Yao model [9]: Cryptographic operations, e.g., E for encryption and D for decryption, are considered as operators in a term algebra where only certain cancellation rules hold. (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.

E. Snekkenes and D. Gollmann (Eds.): ESORICS 2003, LNCS 2808, pp. 271–290, 2003.
© Springer-Verlag Berlin Heidelberg 2003

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 [19]. Closing this gap has motivated a considerable amount of research in security and cryptography over the past few years, e.g., [14,20,3,12,21,8,5].

The abstraction we introduced in [5] 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, memoryless 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 [5], our ideal and real cryptographic libraries were shown to fulfill this definition. The general composition theorem for the underlying model [21] 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 [5], 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 [5]: 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 mes-

sages 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 [5]. Due to lack of space, this paper only briefly sketches the proof; the complete proof can be found in the full version of this paper [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, [11] 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 [21] combine encryption and signatures in a fixed way, while the lower-level encryption subsystem used in that paper, like the examples in [14], does not offer abstract, implementation-independent outputs. The cryptographic primitives in [7,8] 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.

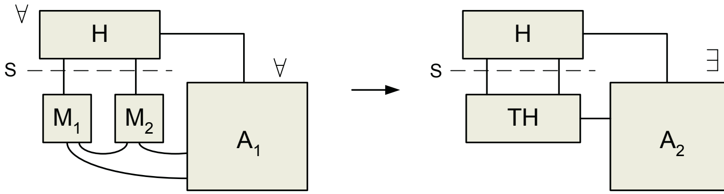

Fig. 1. Overview of the simulatability definition. A real system is shown on the left-hand side, and an ideal system on the right-hand side. The view of H must be indistinguishable.

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 A and honest users summarized as a single machine H. The security definition is that for all polynomial-time honest users H and all polynomial-time adversaries A_1 on the real system, there exists an adversary A_2 on the ideal system such that the honest users H cannot notice the difference, as shown in Figure 1.

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 [5] 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 "$\stackrel{\mathcal{R}}{\leftarrow}$" 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. Each function has to be 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 $u \in \mathcal{H} \cup \{\mathsf{a}\}$. 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 ska, pka, and aut to *typeset* from [5], 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 $\mathsf{p}?$, $\mathsf{TH}_\mathcal{H}$ further 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. The underlying IO automata model guarantees that a machine can enforce such bounds without additional Turing steps even if the adversary tries to send more data. The bounds from [5] can be adopted without modification except that the number of permitted inputs from the adversary has to be enlarged. This is just a technical detail to allow for a correct proof of simulatability. We omit further details and refer to the full version [4].

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 $\mathsf{in}_\mathsf{a}?$; they model tolerated imperfections, i.e., possibilities that an adversary may have, but honest users do not. Send commands output values to other users. In the following, the notation $j \leftarrow \mathsf{algo}(i)$ for a command algo of $\mathsf{TH}_\mathcal{H}$ means that $\mathsf{TH}_\mathcal{H}$ receives an input $\mathsf{algo}(i)$ and outputs j if the input and output port are clear from the context. We only allow lists to be 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 [5] are put into messages.

For dealing with symmetric authentication we 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 based on its abstract state. Handle arguments are tacitly required to be in $\mathcal{HNDS}$ and existing,

i.e., $\leq curhnd_u$, at the time of execution. The underlying model further bounds the length of each input to ensure polynomial runtime; these bounds are not written out explicitly, but can easily be derived from the domain expectations given for the individual inputs.

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

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

$$D :\Leftarrow (ind := size\text{++}, type := \mathsf{pka}, arg := (), len := 0);$$
$$D :\Leftarrow (ind := size\text{++}, type := \mathsf{ska}, arg := (ind - 1), \\ hnd_u := ska^{\mathsf{hnd}}, len := \mathsf{ska_len}^*(k)).$$

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 $aut^{\mathsf{hnd}} := curhnd_u$++, $pka := ska - 1$ and

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

The general argument format for entries of type 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 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].

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

$$D :\Leftarrow (ind := size{++}, type := \mathsf{aut}, arg := (l, pka_1), \\ hnd_{\mathsf{a}} := trans_aut^{\mathsf{hnd}}, len := D[aut].len).$$

– *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 $v := \mathsf{false}$.

[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.

– *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 pka, ska, and aut.

Note that for authenticators, a handle to the "empty" public key is output in 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.

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 [5]; 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 $\mathsf{out}_v!$.

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 sent between machines. The commands are implemented by real cryptographic algorithms, and the simulatability proof will show that nevertheless, everything a real adversary can achieve can also be achieved by an adversary in the ideal system, or otherwise the underlying cryptography can be broken. We now present our additions and modifications to the real system, 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 true (then we say that the authenticator is valid) or 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 ska_len and 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 [10]. We only use our notation for interacting machines, and we allow that also the test function is adaptively attacked.

Definition 1. *(Authentication Security) Given an authentication scheme, an* authentication machine 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* (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* Aut*, the probability is negligible (in k) that* Aut *outputs* $v =$ 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 adv_transform_aut as a tolerable imperfection in Section 3.4. A well-known example of an authentication scheme that is provably secure under this definition is HMAC [6].

4.2 Structures

The intended structure of the real cryptographic library consists of n machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$. Each M_u has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$, so that the same honest users can connect to the ideal and the real system. Each M_u has connections to each M_v exactly as in [5], in particular an insecure connection called $\mathsf{net}_{u,v,\mathsf{i}}$ for normal use. They are called network connections and the corresponding ports network

ports. Any subset $\mathcal{H}$ of $\{1, \ldots, n\}$ can denote the indices of correct machines. The resulting actual structure consists of the correct machines with modified channels according to a channel model. In particular, an insecure channel is split in the actual structure so that both machines actually interact with the adversary. Details of the channel model are not needed here. Such a structure then interacts with honest users H and an adversary A.

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 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 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. 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

[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 pks and pke denoting public signature and encryption keys. No further knowledge of such types is needed for understanding the new work.

returns i^{hnd}, assigning the input values *type* and *add_arg* to the corresponding attributes of $D_u[i^{\mathsf{hnd}}]$ only if $D_u[i^{\mathsf{hnd}}].type$ was null. Else if $|i| > \mathsf{max_len}(k)$, 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 $\mathsf{p}?$, 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 adopted from the original library without modification. Length functions for inputs are tacitly defined by the domains of each input again.

4.5 Inputs and Their Evaluation

Now we describe how M_u evaluates individual new inputs.

Constructors and One-Level Parsing. The stateful commands are defined via functional constructors and parsing algorithms for each type. A general functional algorithm

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

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

“parse m^{hnd}”

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

“parse m^{hnd} if necessary”

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

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

- *Key constructor:* $sk^* \leftarrow \mathsf{make_auth_key}()$.
  Let $sk \leftarrow \mathsf{gen_A}(1^k)$, $sr \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce_len}(k)}$, and return $sk^* := (\mathsf{ska}, sk, sr)$.
- *Key generation:* $ska^{\mathsf{hnd}} \leftarrow \mathsf{gen_auth_key}()$.
  Let $sk^* \leftarrow \mathsf{make_auth_key}()$, $ska^{\mathsf{hnd}} := curhnd_u{+}{+}$, and $D_u :\Leftarrow (ska^{\mathsf{hnd}}, sk^*, \mathsf{ska}, ())$.
- *Key parsing:* $arg \leftarrow \mathsf{parse_ska}(sk^*)$.
  If sk^* is of the form (ska, sk, sr) with $sk \in \{0,1\}^{\mathsf{ska_len}(k)}$ and $sr \in \{0,1\}^{\mathsf{nonce_len}(k)}$, return $()$, else $\downarrow$.
- *Authenticator constructor:* $aut^* \leftarrow \mathsf{make_auth}(sk^*, l)$, for $sk^*, l \in \{0,1\}^+$.
  Set $r \xleftarrow{\mathcal{R}} \{0,1\}^{\mathsf{nonce_len}(k)}$, $sk := sk^*[2]$ and $sr := sk^*[3]$. Authenticate as $aut \leftarrow \mathsf{auth}_{sk}((r, l))$, and return $aut^* := (\mathsf{aut}, sr, r, l, aut)$.

- *Authenticator generation:* $aut^{\mathsf{hnd}} \leftarrow \mathsf{auth}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.
  Parse l^{hnd} if necessary. If $D_u[ska^{\mathsf{hnd}}].type \neq \mathsf{ska}$ or $D_u[l^{\mathsf{hnd}}].type \neq \mathsf{list}$, then return $\downarrow$. Otherwise set $sk^* := D_u[ska^{\mathsf{hnd}}].word$, $l := D_u[l^{\mathsf{hnd}}].word$, and $aut^* \leftarrow \mathsf{make_auth}(sk^*, l)$. If $|aut^*| > \mathsf{max_len}(k)$, return $\downarrow$, else set $aut^{\mathsf{hnd}} := curhnd_u$++ and $D_u :\Leftarrow (aut^{\mathsf{hnd}}, aut^*, \mathsf{aut}, ())$.
- *Authenticator parsing:* $arg \leftarrow \mathsf{parse_aut}(aut^*)$.
  If aut^* is not of the form $(\mathsf{aut}, sr, r, l, aut)$ with $sr, r \in \{0,1\}^{\mathsf{nonce_len}(k)}$, $l \in \{0,1\}^+$, and $aut \in \{0,1\}^{\mathsf{aut_len}'(k,|l|)}$, return $\downarrow$. Also return $\downarrow$ if l is not a tagged list. Otherwise set $arg := (l)$.
- *Authenticator verification:* $v \leftarrow \mathsf{auth_test}(aut^{\mathsf{hnd}}, ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$.
  Parse aut^{hnd} yielding $arg =: (l)$, and parse ska^{hnd}. If $D_u[aut^{\mathsf{hnd}}].type \neq \mathsf{aut}$ or $D_u[ska^{\mathsf{hnd}}].type \neq \mathsf{ska}$, return $\downarrow$. Else let $(\mathsf{aut}, sr, r, l, aut) := D_u[aut^{\mathsf{hnd}}].word$ and $sk := D_u[ska^{\mathsf{hnd}}].word[2]$. If $sr \neq D_u[ska^{\mathsf{hnd}}].word[3]$ or $l \neq D_u[l^{\mathsf{hnd}}].word$, or $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{false}$, output $v := \mathsf{false}$, else $v := \mathsf{true}$.
- *Message retrieval:* $l^{\mathsf{hnd}} \leftarrow \mathsf{msg_of_aut}(aut^{\mathsf{hnd}})$.
  Parse aut^{hnd} yielding $arg =: (l)$. If $D_u[aut^{\mathsf{hnd}}].type \neq \mathsf{aut}$, return $\downarrow$, else let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

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 $v \in \{1, \dots, n\}$, M_u parses l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$, M_u outputs $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,\mathsf{i}}!$.

Inputs at network ports are simply tested for being tagged lists and stored as in [5].

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 A, the combination $\mathsf{Sim}_{\mathcal{H}}(\mathsf{A})$ of $\mathsf{Sim}_{\mathcal{H}}$ and A achieves the same effects in the ideal system as the adversary A in the real system, 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_{a} and a variable $curhnd_{\mathsf{a}}$. Each entry in D_{a} has the following attributes:

- $x.hnd_{\mathsf{a}} \in \mathcal{HNDS}$ is used as the primary key attribute in D_{a}. However, its use is not as straightforward as in the ideal and real system, since entries are created by completely parsing an incoming message recursively.

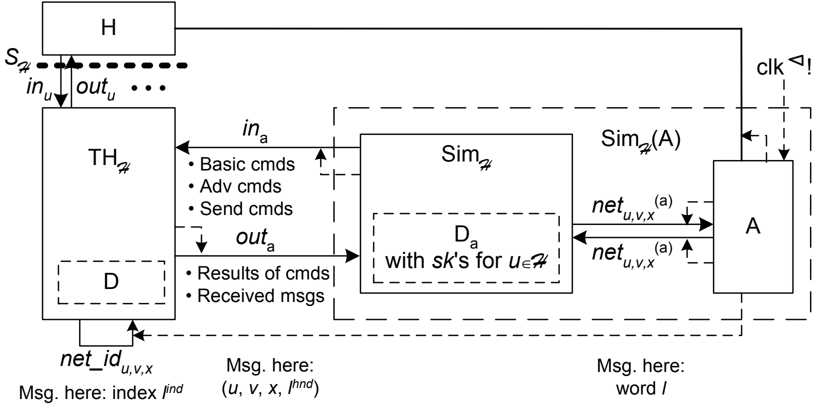

Fig. 2. Set-up of the simulator.

– $x.word \in \{0,1\}^*$ is the real representation of x.
– $x.add_arg$ is a list of additional arguments. Typically it is (). However, for our key identifiers it is (adv) if the corresponding secret key was received from the adversary, while for keys from honest users, where the simulator generated an authentication key, it is of the form (honest, sk^*).

The variable $curhnd_{\mathsf{a}}$ denotes the current size of D_{a}, except temporarily within an algorithm id2real. Similar to $\mathsf{TH}_{\mathcal{H}}$, the simulator maintains a counter $steps_{\mathsf{p}?} \in \mathbb{N}_0$ for each input port p? for the number of inputs at that port, initialized with 0. Similar to the definition of $\mathsf{TH}_{\mathcal{H}}$, the corresponding bounds $bound_{\mathsf{p}?}$ can be adopted one-to-one from [5], with the only exception that the bound for $\mathsf{out_a}$? has to be enlarged.

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 $\mathsf{net}_{u,v,\mathsf{i}}!$.

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^{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 (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^{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^{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_{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 $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ 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 $D_{\mathsf{a}} :\Leftarrow (curhnd_{\mathsf{a}}{+}{+}, \epsilon, (\mathsf{adv}))$ (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^{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 [5], except that the interaction of $\mathsf{TH}_{\mathcal{H}}$ and $\mathsf{Sim}_{\mathcal{H}}$ in $\mathsf{real2id}$ 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 bounds at $\mathsf{in_a}?$ and $\mathsf{out_a}?$ 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, where $\mathsf{max_in}(k)$ denotes the permitted number of inputs for each honest user, plus the polynomial runtime of A for the remaining ones). We omit further details.

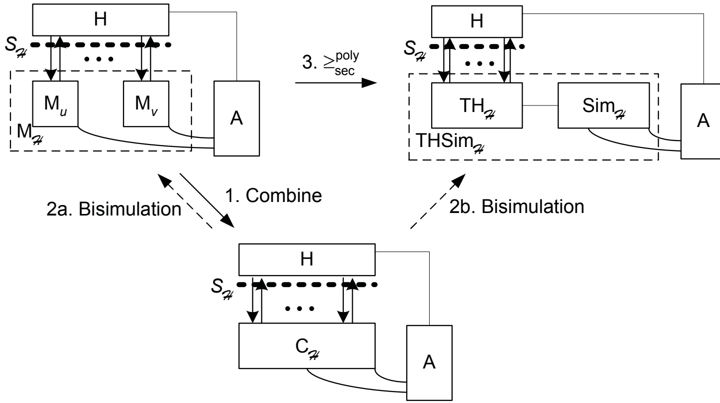

Fig. 3. Overview of the Simulatability Proof.

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 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. As the rigorous proof of correct simulation takes 11 pages, we only give a brief sketch 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. We need a probabilistic bisimulation because the real system and the simulator are probabilistic , i.e., identical outputs should yield mapped states with the correct probabilities and identically distributed outputs.

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.

In addition to standard invariants, we have an information-flow invariant 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.

Finally, the adversary might succeed in attacking the real system with very small probability, which is impossible in the ideal system. We collect these cases in certain "error sets", and we show that these sets have negligible probability (in the security parameter) at the end; this is sufficient for computational indistinguishability.

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. Symmetric authentication within a simulatable cryptographic library. IACR Cryptology ePrint Archive 2003/145, July 2003. `http://eprint.iacr.org/`.
5. M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003. `http://eprint.iacr.org/`.
6. 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.
7. R. Canetti. A unified framework for analyzing security of protocols. IACR Cryptology ePrint Archive 2000/067, Dec. 2001. `http://eprint.iacr.org/`.
8. 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.
9. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
10. 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.
11. 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.
12. M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
13. R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

14. 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.
15. 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.
16. C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.
17. J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.
18. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
19. 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`.
20. 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.
21. 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.
22. 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.
23. S. Schneider. Security properties and CSP. In *Proc. 17th IEEE Symposium on Security & Privacy*, pages 174–187, 1996.