International Journal on Information Security manuscript No.
(will be inserted by the editor)

Symmetric Authentication in a Simulatable Dolev-Yao-style Cryptographic Library

Michael Backes, Birgit Pfitzmann, Michael Waidner

IBM Zurich Research Lab
e-mail: {mbc,bpf,wmi}@zurich.ibm.com

Abstract Tool-supported proofs of security protocols typically rely on abstractions from real cryptography by term algebras, so-called Dolev-Yao models. However, until recently it was not known whether a Dolev-Yao model can be implemented with real cryptography in a provably secure way under active attacks. For public-key encryption and signatures, this was recently shown, if one accepts a few additions to a typical Dolev-Yao model such as an operation that returns the length of a term.

Here we extend this Dolev-Yao-style model, its realization, and the security proof to include a first symmetric primitive, message authentication. This adds a major complication: we must deal with the exchange of secret keys. For symmetric authentication, we can allow this at any time, before or after the keys are first used for authentication, while working only with standard cryptographic assumptions.

Key words Dolev-Yao, computational soundness, symmetric authentication

1 Introduction

Proofs of security protocols typically employ simple abstractions of cryptographic operations, so that large parts of the proofs become independent of cryptographic details such as polynomial-time restrictions, probabilistic behavior and error probabilities. This is particularly true for tool-supported proofs, e.g., [29,25,19,35,36,1,23,30].

The typical abstraction style is the Dolev-Yao model, or better "models". Cryptographic operations, e.g., E for encryption and D for decryption, are treated 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. This was introduced for unary operations in [11], with first extensions to more general terms and equations in [12, 26, 13].

However, there was traditionally no cryptographic justification, i.e., no theorem that said what a proof with a Dolev-Yao abstraction implied for the real implementation, even if provably secure cryptographic primitives are used. In fact, one can construct at least artificial protocols that are secure in certain Dolev-Yao models, but become insecure if implemented with provably secure cryptographic primitives [31]. Closing this gap has motivated a considerable amount of research over the past few years. The first security proof for a Dolev-Yao-style model under active attacks was presented in [6]. This proven model contains public-key encryption and signatures as the main cryptographic operations, together with nonces, payload messages, and a list operation. Probabilistic term generation is abstracted by a counting mechanism, one of the usual abstractions in prior Dolev-Yao models in particular for nonces, probably first in [25]. The main unusual feature is an operation that returns the length of a term, corresponding to the length in bits of the corresponding real message; this was necessary because encryption cannot hide the message length completely. The implementation uses arbitrary encryption and signature schemes secure under the standard cryptographic definitions for active attacks (adaptive chosen ciphertext and adaptive chosen message, respectively, but not adaptive corruptions); those obtain a few non-cryptographic additions like type tags and additional random message fields. The security holds in the sense of reactive simulatability. Essentially this means that anything an adversary can achieve in a real system can also be achieved by an adversary against the corresponding ideal system, here the Dolev-Yao-style model. The trust model in this simulatability, corresponding to the trust model for the primitives, is adaptive active attacks but non-adaptive corruptions. This is also the prevelant trust model in Dolev-Yao models. Reactive simulatability guarantees arbitrary composability, and typical security properties are preserved from ideal systems to real systems. Composability and property preservation together mean that properties of cryptographic protocols can be proved symbolically over the Dolev-Yao-style model and automatically carry over to the real cryptographic implementation.

However, a limitation in [6] is that only asymmetric cryptographic primitives are considered. For asymmetric primitives it is reasonable to define the Dolev-Yao terms such that secret keys are only used for decryption and signing, but not included in terms in other ways. This restriction does not exclude many typical protocols, and has been used in Dolev-Yao models before – some models even assume completely predistributed public keys. For symmetric primitives, however, such a restriction would be unreasonable: A secret key in a protocol almost always has to be shared by at least two parties, and key-exchange protocols are among those most commonly analyzed with Dolev-Yao models.

The main contribution of this paper is to add the first symmetric primitive to the framework of [6]: message authentication. In other terminologies, this is called keyed hashing or MACs (message authentication codes). In many Dolev-Yao models symmetric authentication is not really distinguished from encryption, or only the instantiations using hash functions are considered, but in cryptography sym-

metric authentication is an important primitive with definitions and constructions of its own.

The inclusion of a symmetric primitive and the sending of secret keys adds a major complication compared with the proof in [6] because a key may be sent at any time before or after it is first used to authenticate a message. In particular, this implies that a real adversary can send a message (bit string) which cannot immediately be represented by a known Dolev-Yao term, because the key needed to test the validity of an authenticator is not yet known, but may be sent later by the adversary. When only public keys are exchanged, the problem can be avoided by tagging all real messages with the public keys used in them, so that messages can immediately be classified into correct terms or a specific garbage type [6].

Defining a reactively secure Dolev-Yao-style abstraction of symmetric authentication as an addition to the model from [6], instead of on its own, has the advantage that we can then consider terms that use both public-key and secret-key operations, e.g., a public-key-encrypted key for symmetric authentication. Further, we thus obtain that all inequalities and non-derivability results that the Dolev-Yao model postulates between such mixed terms carry over to a realization and can therefore be used in symblic protocol proofs. However, this comes at a price: We must define and prove the new operations within the existing framework. This framework has relatively clear extension points for such additions, but nevertheless the overall complexity increases somewhat with every new system.

Related Work. Abadi and Rogaway started to bridge the abstraction gap for Dolev-Yao models [3]. However, they only handled passive adversaries and symmetric encryption. The protocol language and security properties were extended in [2,21], but still only for passive adversaries. This excludes most of the typical ways of attacking protocols, e.g., man-in-the-middle attacks and attacks by reusing a message part in a different place or concurrent protocol run. A justification for arbitrary active attacks and within the context of arbitrary surrounding interactive protocols was first given in [6]. Based on the specific Dolev-Yao model proved there, the well-known Needham-Schroeder-Lowe protocol was symbolically proved in [4]. This shows that in spite of adding certain operators and rules compared with simpler Dolev-Yao models, such a proof is possible in the style already used in automated tools, only now with a sound cryptographic basis. Subsequently, several papers presented cryptographic underpinnings of Dolev-Yao models under active attacks for specific primitives, e.g., [22] for symmetric encryption and [18,28] for public-key encryption.

The security notion of reactive simulatability, a notion of secure implementation that allows arbitrary composition, was first defined generally in [32], based on simulatability definitions for (one-step) function evaluation [15,16,7,27,9]. It was extended in [33,10] and has since been used in many ways for proving individual cryptographic systems and general theorems.

Overview of this Paper. In Section 2, we give an informal overview of the Dolev-Yao-style model of symmetric authentication and show how major concepts, both new ones from this paper and underlying ones from [6], relate to other Dolev-Yao

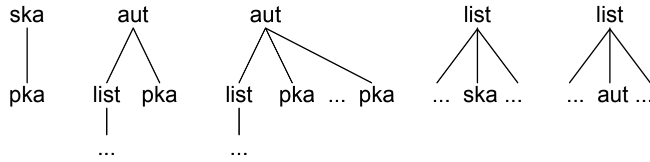


Fig. 1 Term structures with symmetric authentication.

models. Note that while all term algebras are similar, other aspects like interaction, protocol definition, and adversary capabilities are treated in syntactically very diverse ways in the literature. After briefly introducing general notation in Section 3, we rigorously define our Dolev-Yao-style abstraction in Section 4. The realization of this specific abstraction is defined in Section 5. Section 6 describes the security notion of reactive simulatability in more detail than the introduction. Sections 7 and 8 contain the proof. Readers most interested in the proof technique can read the start of these sections before the ideal and real system; however, as the overall technique is the same as in [6], we abbreviate the general parts where possible.

2 Overview of the Dolev-Yao-Style Model of Symmetric Authentication

For modeling and proving cryptographic protocols, it is sufficient to know the ideal, Dolev-Yao-style model of symmetric authentication. The subsequent sections only justify that – and how – the ideal version can be faithfully implemented by real cryptographic primitives fulfilling normal cryptographic definitions. In this section, we give an overview of the Dolev-Yao-style model of symmetric authentication that we will later prove to be securely implementable, and describe the reasons for some major design decisions. This section also motivates some underlying concepts from [6] in the hope that at least readers who know other Dolev-Yao models can later easily map our representation into their favorite one and concentrate on the main issues.

2.1 Terms and Operations

A summary of the types of terms arising by adding symmetric authentication to our Dolev-Yao-style model is shown in Figure 1. At first glance, one might expect just leaves for secret authentication keys, denoted by the type ska, and authenticators of type aut with a message (here represented as list) and a secret key as arguments. A major deviation from this expectation is the type pka, which denotes public tags for authentication keys. This models that the adversary might be able to determine the key that was used to authenticate a message. A secret-key term has such a public tag as an argument, and an authenticator has the public tag of the respective secret key as an argument instead of that secret key. This explains the first and second term template in Figure 1. The next deviation from what one might expect is that there are authenticators with any number $j \in \mathbb{N}_0$ of keys (i.e., their public tags) as arguments instead of one. Such terms can be produced by the adversary

either by sending an authenticator first, so that the key is not yet known ($j = 0$) or by finding keys such that authenticators are valid with respect to several keys. This is not excluded by typical cryptographic security definitions and indeed does no harm in typical uses of authentication. Finally, one might expect a test operator, but Dolev-Yao models exist with and without destructors, i.e., operations like decryption and signature testing may be explicit elements in terms (as in the introduction) or implicit. The version in [6] belongs to the second class. Thus an application of a test to a term is always immediately evaluated. The last two term templates in Figure 1 show that secret keys and authenticators may be included in lists and thus in many larger terms.

The honest participants operate on these terms in the expected ways: They can generate keys, authenticate messages (yielding normal terms with one key argument), test authenticators, and we allow them to extract messages from authenticators (i.e., the second and third term in Figure 1 allow retrieval of the list). This last operation 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 both explicitly and implicitly.

The adversary has a few additional possibilities: As already mentioned, he may construct authenticator terms with $j \neq 1$ public key tags, also by adding key tags to authenticators originally constructed by an honest user, and he can extract key tags from authenticators.

2.2 Abstraction from Probabilism and Participant Knowledge

Like all Dolev-Yao-style models when actually used for protocol modeling, e.g., using a special-purpose calculus or embedded in CSP or pi-calculus, the model in [6] has state. An important use of state is to model which participants already know which terms. Another use of state is to remember different versions of terms of the same structure for probabilistic operations such as nonce or key generation. We allow probabilistic authentication; hence also the authentication operation generates a new version of an authenticator term at each call. In [6], as in some prior models – probably first in [25] – the probabilism is abstracted from by counting, i.e., by assigning successive natural numbers to terms, here globally over all types.

A specific aspect in [6] is that participants operate on terms by local names, not by handling the terms directly. This is necessary to give the abstract Dolev-Yao-style model and its realization the same interface, so that either one or the other can be plugged into a protocol. An identical interface is also an important precondition for reactive simulatability, i.e., the security notion. One can see protocol descriptions over this interface as a low-level symbolic representation as they exist in several other frameworks, and it should be possible to compile higher-level descriptions into it following the ideas first developed in [24]. The local names are called handles, and chosen as successive natural numbers for simplicity.

The handles also implicitly define the knowledge sets of other models: The knowledge set of a participant, including the adversary, is the set of terms for which this participant has handles.

3 Notation

We write ":=" for deterministic and "$\leftarrow$" for probabilistic assignment, and "$\leftarrow_R$" 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 $D :\Leftarrow x$.

4 The Dolev-Yao-Style Model

We now present our abstraction from symmetric authentication in detail. Before we can rigorously define the new terms and the operations on terms and the overall state, e.g., on the handles representing knowledge sets, we have to introduce notation from the overall model from [6] to which we add these terms and operations.

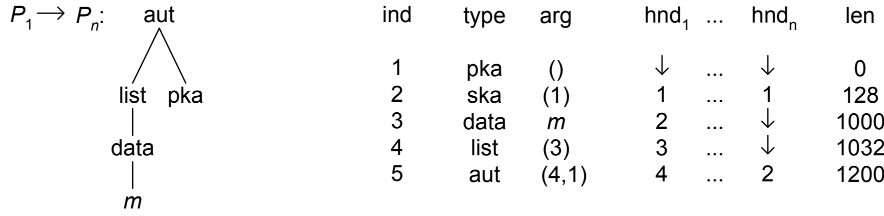
4.1 Trusted-Host Machines and Overall Parameters

The underlying system model is an IO-automata model. Hence the overall Dolev-Yao-style model, with its state, is represented as a machine. It is called trusted host. Actually there is one possible 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 trusted host keeps track of the length of messages (recall that this is needed because the length leaks to the adversary) 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. They follow the same conventions as the other functions in [6], in particular they range over $\mathbb{N}$, are polynomially bounded, and efficiently computable. Another two functions from L that we need below are $\mathsf{max_in}(k)$ and $\mathsf{max_in_a}(k)$. They can be arbitrary polynomials and denote the maximum number of inputs at each user port and at the adversary port, respectively.

4.2 States: Term Database

The overall representation of a state of the Dolev-Yao-style model, i.e., of the machine $\mathsf{TH}_\mathcal{H}$, is a database D of the existing terms with their type (top-level



ind	type	arg	hnd$_1$	...	hnd$_n$	len
1	pka	()	↓	...	↓	0
2	ska	(1)	1	...	1	128
3	data	*m*	2	...	↓	1000
4	list	(3)	3	...	↓	1032
5	aut	(4,1)	4	...	2	1200

Fig. 2 Example of the database representation of terms.

operator), argument list, handles, and lengths as database attributes. In addition, it contains a global index that allows us (not the participants) to refer to terms unambiguously. The non-atomic arguments of a term are given by the indices of the respective subterm.

An example is shown in Figure 2. The left side indicates the main action that has happened so far, the sending of an authenticated list with one element, a payload message m. The database first contains the key pair, where the secret key is supposed to be known to both participants, while honest participants never have special handles to the public key tags. Then it contains the payload data, the list, and the authenticated message. We assume that this message has arrived safely so that P_n has a handle to it, but has not yet been parsed by the recipient. After that, the list and m get handles 3 and 4 for P_n, respectively.

In detail, the database attributes of D are defined as follows; the only difference to [6] due to adding symmetric authentication is an augmented type set.

- $ind \in \mathcal{INDS}$, 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 $D[ind = i]$.
- $type \in typeset$ defines the type of the entry. We add types ska, pka, and aut to $typeset$ from [6], denoting secret authentication keys, "empty" public keys that are needed as key identifier for the corresponding secret keys, and authenticators. The type pka is added to the subset $secrettypes \subseteq typeset$, which consists of those types that must not be put into lists (formerly only consisting of the secret keys of asymmetric schemes).
- $x.arg = (a_1, a_2, \ldots, a_j)$ 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. The value $\downarrow$ 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, computed using the functions from L.

Initially, D is empty. As additional state parts, $\mathsf{TH}_{\mathcal{H}}$ has a counter $size \in \mathcal{INDS}$ for the current size of D, and counters $curhnd_u$ (current handle) for $u \in$

$\mathcal{H} \cup \{\mathsf{a}\}$, denoting the most recent handle number assigned for u. They are all initialized with 0.

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.

For each input port 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. 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 is $\mathsf{max_in_a}(k)$.

4.3 New Inputs and their Evaluation

According to the underlying IO-automata model, operations are represented by input commands from users or the adversary into $\mathsf{TH}_{\mathcal{H}}$. Thus the use of this Dolev-Yao-style model is very much like the use of a real cryptographic library by an application or protocol implementation, referring to cryptographic objects by object handles. The normal cryptographic operations are called basic commands. They are accepted at input ports in_u?; they correspond to cryptographic operations and have only local effects, i.e., only an output at out_u? occurs and only handles for u are involved. The additional term-handling capabilities of the adversary are called local adversary commands. They are only accepted at $\mathsf{in_a}$?. The last group, called send commands, output values to other users. The normal insecure channels actually lead to the adversary, i.e., the adversary instead of the intended recipient gets a handle on a sent message, and the adversary can send under anyone's identity.

In the following, the notation $j \leftarrow \mathsf{op}(i)$ means that $\mathsf{TH}_{\mathcal{H}}$ is scheduled with an input $\mathsf{op}(i)$ at some port in_u? (where we always use u as the index of that port) and returns j at out_u!. The definitions in [6] 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 are put into messages. Handle arguments are tacitly required to be in $\mathcal{HNDS}$ and existing, i.e., $\leq curhnd_u$, at the time of execution.

4.3.1 Basic Commands: Normal Cryptographic Operations As introduced in Section 2.1 there are four local cryptographic operations for all participants.

Definition 1 *(Basic commands for symmetric authentication) The trusted host* $\mathsf{TH}_{\mathcal{H}}$ *extended by symmetric authentication accepts the following additional commands at every ports* in_u?.

– *Key generation:* $ska^{\mathsf{hnd}} \leftarrow \mathsf{gen_auth_key}()$. *Set* $ska^{\mathsf{hnd}} := curhnd_u{+}{+}$ *and*

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

– *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*

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

– *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* $v := \mathsf{true}$.
– *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* $l^{\mathsf{hnd}} := \mathsf{ind2hnd}_u(l)$.

The tests in authenticator generation are input type checks and a test that the resulting message will not exceed the given polynomial bound. (The latter is just a provability issue; the bound should be so large as to be never reached in practice.)

In verification, the test $ska-1 \notin \{pka_1, \ldots, pka_j\}$ 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.

4.3.2 Local Adversary Commands We already discussed in Section 2.1 that we allow the adversary to send terms with authenticators for which it has not sent a suitable key yet. We call such authenticators (temporarily) unknown. Such an authenticator can become valid if a suitable secret key is received; a command for fixing authenticators takes care of this. In addition, we allow the adversary to transform an authenticator, i.e., create a new authenticator for a message if he already knows another authenticator for the same message. This capability is not excluded by typical security definitions. Finally, we allow the adversary to retrieve all information that we do not explicitly require to be hidden, e.g., type and arguments of a term with a given handle.

Definition 2 *(Local adversary commands for symmetric authentication) The trusted host* $\mathsf{TH}_{\mathcal{H}}$ *extended by symmetric authentication accepts the following additional commands at the port* $\mathsf{in_a}?$.

– *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*

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

– *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* pka, ska, *and* aut.

The fact that adv_parse applied to an authenticator outputs a handle to the public key models that the adversary might be able to see which authenticators were made with the same key. By itself, such a public key is meaningless, but the adversary can compare the public keys from different authenticators.

4.3.3 Send Commands, Unchanged The send commands are unchanged by adding symmetric authentication. However, they are needed in the proof because simulation only happens when terms are sent or received. Hence as an example we present sending over an insecure channels (denoted by a parameter i). This is the most commonly used type, but there are also secure channels and authentic channels. Essentially, sending increases knowledge sets and thus assigns and outputs handles. Intuitively, in the first command an honest user wants to send list l to user v. In the second command, the adversary wants to send list l to v, pretending to be u.

– $\mathsf{send_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$. 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 $\mathsf{in_a}?$. 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 $\mathsf{out}_v!$.

M_1

hnd_1	type	word	add_arg
1	ska	x4lg49m...	()
2	data	m	()
3	list	(m)	()
4	aut	(m, pb33qy...)	()

M_n

hnd_n	type	word	add_arg
1	ska	x4lg49m...	()
2	null	(m, pb33qy...)	()

Fig. 3 Real situation for the same example as above.

5 Real System

The realization of the Dolev-Yao-style model offers its users the same interface as the ideal model, i.e., honest users operate on cryptographic objects via handles. Clearly, in the real system every user has its own machine (in other terms protocol engine or automaton), containing only the cryptographic objects that this user knows. Figure 3 shows the real situation corresponding to the example from Figure 2. Essentially, the machines of user P_1 and P_n contain the projections of the Dolev-Yao-style database to the objects for which this user has handles, with terms replaced by bitstrings. In the example bitstrings, we represent a list with brackets, and an authenticator (which allows message retrieval) as a pair of the message and a basic authenticator. As user P_n has not yet parsed the authenticator, it still has the type null in machine M_n, see below.

The commands like key generation and authentication essentially call the underlying cryptographic algorithms; however, we need some additional tagging and randomization. Upon send commands, these machines exchange the real bitstrings over real channels. The adversary can arbitrarily manipulate the messages on insecure channels and in his local knowledge, i.e., perform any polynomial-time algorithms on the bitstrings.

We start the rigorous treatment with the underlying definitions of a cryptographically secure symmetric authentication system.

5.1 Cryptographic Definition of Symmetric Authentication

The following is a standard definition of authentication codes except that we require that all algorithms have fixed effects on the parameter lengths. This can easily be achieved by padding, given any other authentication code.

Definition 3 *(Memoryless symmetric authentication) A memoryless symmetric authentication scheme is a tuple* $\mathcal{A} = (\mathsf{gen_A}, \mathsf{auth}, \mathsf{atest}, \mathsf{ska_len}, \mathsf{aut_len})$ *of polynomial-time algorithms. For key generation with a security parameter* $k \in \mathbb{N}$*, we write* $sk \leftarrow \mathsf{gen_A}(1^k)$*. 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*.*

With these parameter notation, the length of sk *must always be* $\mathsf{ska_len}(k) > 0$*. Correctly generated authenticators for keys of the correct length must always be*

valid. The length of aut must be $\mathsf{aut_len}(k, |m|) > 0$*, and this is also the length of 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 [17]. We only use our notation for interacting machines, and we allow that also the test function is adaptively attacked.

Definition 4 *(Authentication Security) Given an authentication scheme, an* authentication machine Aut *has one input and one output port, a variable sk initialized as $sk \leftarrow \mathsf{gen_A}(1^k)$, and the following transition rules:*

- *On input* (auth, m)*, return* $aut \leftarrow \mathsf{auth}_{sk}(m)$.
- *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* $v = \mathsf{true}$ *on any input* $(\mathsf{test}, aut', m')$ *where m' was not authenticated until that time, i.e., not among the inputs called m.*

The definition does not exclude that the adversary constructs another authenticator $aut' \neq aut$ for a message m that was authenticated. This is why we introduced the command $\mathsf{adv_transform_aut}$ in Section 4.3.2. A well-known example of an authentication scheme that is provably secure under this definition is HMAC [8].

5.2 Machines and Parameters

The intended structure of the realization consists of n machines $\{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$, one for each participant. Each M_u has ports $\mathsf{in}_u?$ and $\mathsf{out}_u!$, so that the same honest-user machines (representing applications or protocols) can connect to the ideal and the real system. Each M_u has connections to each M_v as in [6], 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, each insecure channel is split so that both machines actually interact with the adversary.

Similarly to the length functions $\mathsf{ska_len}$ and $\mathsf{aut_len}$ from the cryptographic definition, there are underlying functions $\mathsf{list_len}$ and $\mathsf{nonce_len}$ defining the length of lists (based on the element lengths) and nonces. These functions are grouped in a tuple L' and 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 $\mathsf{max_len}(k)$;

– $\mathsf{aut_len}'(k, l) := \mathsf{aut_len}(k, \mathsf{list_len}(\mathsf{nonce_len}(k), l))$;
– $\mathsf{aut_len}^*(k, l) := \mathsf{list_len}(|\mathsf{aut}|, \mathsf{nonce_len}(k), \mathsf{nonce_len}(k), l, \mathsf{aut_len}'(k, l))$.

This is mainly needed to relate this real system to an ideal system with suitable parameters, but we use one of these definitions already in defining the real system.

5.3 States of a Machine

Each machine M_u contains the cryptographic objects the user already knows under this user's handles. We again represent this as a database; the structure is simple here because there are only the handles, the corresponding bitstrings, and for our convenience the message types (which could be retrieved by parsing the bitstring) and a "reserve" parameter. More precisely, each such database D_u has the following attributes:

– $hnd_u \in \mathcal{HNDS}$ 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 $D_u[hnd_u = i^{\mathsf{hnd}}]$.
– $word \in \{0,1\}^+$ is the real bitstring.
– $type \in typeset \cup \{\mathsf{null}\}$ identifies the type of the entry. The value null denotes that the entry has not yet been parsed.
– add_arg is a list of ("additional") arguments. For entries of our new types it is always empty, i.e., $()$.

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., $i^{\mathsf{hnd}} := D_u[word = i \wedge type \notin \{\mathsf{sks}, \mathsf{ske}\}].hnd_u \neq \downarrow$,[1] it returns i^{hnd}, assigning the input values $type$ and add_arg to the corresponding attributes of $D_u[i^{\mathsf{hnd}}]$ only if $D_u[i^{\mathsf{hnd}}].type$ was null. Else if $|i| > \mathsf{max_len}(k)$, it returns $i^{\mathsf{hnd}} = \downarrow$. Otherwise, it sets and returns $i^{\mathsf{hnd}} := curhnd_u{++}$, $D_u :\Leftarrow (i^{\mathsf{hnd}}, i, type, add_arg)$.

Similar to Section 4.2, M_u maintains a counter $steps_{\mathsf{p?}} \in \mathbb{N}_0$ for each input port p?, initialized with 0. All corresponding bounds $bound_{\mathsf{p?}}$ are $\mathsf{max_in}(k)$. Length functions for inputs are tacitly defined by the domains of each input.

5.4 Inputs and their Evaluation

Now we describe how M_u evaluates the individual inputs related to symmetric authentication. Clearly, there are the same four basic commands corresponding to

[1] 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 model from [6]. 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.

cryptographic operations. There are no adversary local commands because a real adversary is not restricted to specific, algebraic operations, but performs arbitrary bitstring manipulations. The send commands now correspond to real sending, and receiving a message from a channel into a real machine must newly be considered.

5.4.1 Constructors and One-level Parsing The stateful commands are defined via functional constructors and parsing algorithms for each cryptographic type. (These stateless algorithms can be reused in the simulator and the proof, while the stateful parts are different in the simulator.) We start with the constructors; they define the exact structure of the bitstrings in the database.

Definition 5 *(Constructors for symmetric authentication)*

- *Key constructor:* $sk^* \leftarrow \mathsf{make_auth_key}()$.
  Let $sk \leftarrow \mathsf{gen_A}(1^k)$*,* $sr \leftarrow_R \{0,1\}^{\mathsf{nonce_len}(k)}$*, and return* $sk^* := (\mathsf{ska}, sk, sr)$.
- *Authenticator constructor:* $aut^* \leftarrow \mathsf{make_auth}(sk^*, l)$*, for* $sk^*, l \in \{0,1\}^+$.
  Set $r \leftarrow_R \{0,1\}^{\mathsf{nonce_len}(k)}$*,* $sk := sk^*[2]$ *and* $sr := sk^*[3]$. *Authenticate as* $aut \leftarrow \mathsf{auth}_{sk}((r,l))$*, and return* $aut^* := (\mathsf{aut}, sr, r, l, aut)$.

Now we define the destructors. Authenticator parsing does not include the verification test; that must be defined in the stateful part. The term “tagged list” means a valid list of the real system. We assume that tagged lists are efficiently encoded into $\{0,1\}^+$. From the underlying Dolev-Yao-style model, we need to know the general parsing algorithm.

- *General parsing:* $(type, arg) \leftarrow \mathsf{parse}(m)$.
  If m is not of the form $(type, m_1, \ldots, m_j)$ with $type \in typeset \setminus \{\mathsf{pka}, \mathsf{sks}, \mathsf{ske}, \mathsf{garbage}\}$ and $j \geq 0$, returns $(\mathsf{garbage}, ())$. Else call the type-specific parsing algorithm $arg' \leftarrow \mathsf{parse_}type(m)$. If $arg = \downarrow$, then parse again outputs $(\mathsf{garbage}, ())$, else $(type, arg)$.

The destructors for the symmetric authentication types contain the appropriate type-specific parsing subroutines.

Definition 6 *(Destructors for symmetric authentication)*

- *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 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* $arg := (l)$.

5.4.2 Realization of Basic Commands We now define how a real machine reacts on the same basic commands as the ideal Dolev-Yao-style system. They are again local. In the real system this means that they produce no outputs at the network ports. We use the functional subroutines defined above, and subroutines for the state changes resulting from parsing, defined in [6]:

- "parse m^{hnd}" means 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.
- "parse m^{hnd} if necessary" means the same except that M_u does nothing if $D_u[m^{\mathsf{hnd}}].type \neq \mathsf{null}$.

Definition 7 *(Basic commands for symmetric authentication)*

- *Key generation:* $ska^{\mathsf{hnd}} \leftarrow \mathsf{gen_auth_key}()$.
  Let $sk^* \leftarrow \mathsf{make_auth_key}()$*,* $ska^{\mathsf{hnd}} := curhnd_u{++}$*, and* $D_u :\Leftarrow (ska^{\mathsf{hnd}}, sk^*, \mathsf{ska}, ())$.
- *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 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* $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$.

5.4.3 Send Commands and Network Inputs The send commands are again not specific to symmetric authentication, but as an example we show what happens for an insecure channel.

- $\mathsf{send_i}(v, l^{\mathsf{hnd}})$, for $v \in \{1, \ldots, n\}$.
  Parse l^{hnd} if necessary. If $D_u[l^{\mathsf{hnd}}].type = \mathsf{list}$, output $D_u[l^{\mathsf{hnd}}].word$ at port $\mathsf{net}_{u,v,\mathsf{i}}!$.

An input at a network port should be tagged list. If it is, it is stored under a handle, and the arrival of the message is indicated to the user.

- *Network input:* On input l at $\mathsf{net}_{w,u,\mathsf{i}}?$, for $l \in \{0,1\}^+$ and $|l| \leq \mathsf{max_len}(k)$.
  Test if $l = (\mathsf{list}, x_1, \ldots, x_j)$ for some $j \in \mathbb{N}_0$ and values $x_i \in \{0,1\}^+$. If yes, let $(l^{\mathsf{hnd}}, D_u) :\leftarrow (l, \mathsf{list}, ())$ and output (w, x, l^{hnd}) at $\mathsf{out}_u!$.

6 Definition of Simulatability

We give the definition of the underlying security notion of reactive simulatability. It is intended for comparing an ideal and a real system with respect to security. Generally, an ideal or real system may consist of several possible structures, typically derived from an intended structure with a trust model, where each such structure consists of a set of machines and a set of so-called service ports. For example, the

Dolev-Yao-style model from Section 4 consists of structures $(\{\mathsf{TH}_{\mathcal{H}}\}, S_{\mathcal{H}})$, one for each $\mathcal{H}$, where $S_{\mathcal{H}} := \{\mathsf{in}_u?, \mathsf{out}_u! \mid u \in \mathcal{H}\}$. A structure is complemented to a *configuration* by honest users summarized as a single machine H and an adversary A. H connects only to the service ports of the structure and A to the rest, and they may interact. The set of configurations of a system Sys is called $\mathsf{Conf}(Sys)$. A configuration is a runnable scenario, i.e., for each value of the security parameter k one gets a well-defined probability space of *runs*. The *view* of a machine in a run is the restriction to all in- and outputs this machine sees and its internal states. Formally, the view $view_{conf}(\mathsf{M})$ of a machine M in a configuration $conf$ is a *family of random variables* with one element for each value k of the security parameter.

The security definition for comparing an ideal system Sys_{id} and a real system Sys_{real} is that for every structure $(\hat{M}_1, S) \in Sys_{\mathsf{real}}$, every polynomial-time honest user H, and every polynomial-time adversary A_1, there exists a polynomial-time adversary A_2 on an ideal structure $(\hat{M}_2, S) \in Sys_{\mathsf{id}}$ with the same service ports such that the view of H is computationally indistinguishable in the two configurations, i.e., such that the honest users H cannot notice the difference. This is illustrated in Figure 4. Indistinguishability is a well-known cryptographic notion from [37].

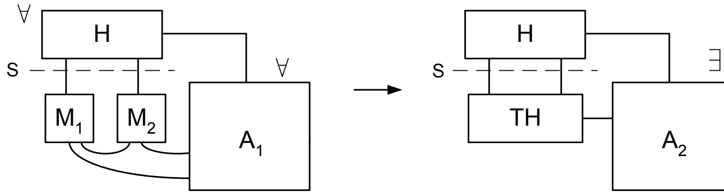


Fig. 4 Overview of the simulatability definition. A real system is shown on the left-hand side, and an ideal system on the right-hand side. The view of H must be indistinguishable.

Definition 8 *(Computational Indistinguishability) Two families* $(\mathsf{var}_k)_{k\in\mathbb{N}}$ *and* $(\mathsf{var}'_k)_{k\in\mathbb{N}}$ *of random variables on common domains* D_k *are* computationally indistinguishable *("≈") iff for every algorithm* D *(the distinguisher) that is probabilistic polynomial-time in its first input,*

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

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

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

Definition 9 *(Reactive Simulatability) For two systems* Sys_{real} *and* Sys_{id}*, one says* $Sys_{\mathsf{real}} \geq Sys_{\mathsf{id}}$ *(*at least as secure as*) iff for every polynomial-time configuration*

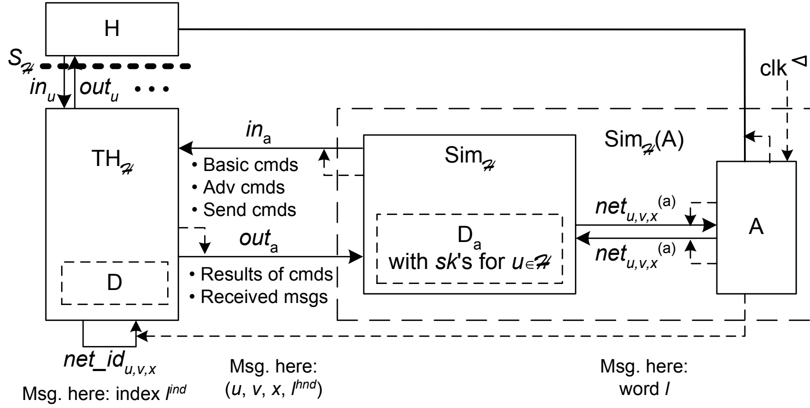


Fig. 5 Environment of the simulator.

$conf_1 = (\hat{M}_1, S, \mathsf{H}, \mathsf{A}_1) \in \mathsf{Conf}(Sys_{\mathsf{real}})$, *there exists a polynomial-time configuration* $conf_2 = (\hat{M}_2, S, \mathsf{H}, \mathsf{A}_2) \in \mathsf{Conf}(Sys_{\mathsf{id}})$ *(with the same* H*) such that* $view_{conf_1}(\mathsf{H}) \approx view_{conf_2}(\mathsf{H})$.

For the proof in [6], this is even shown with blackbox simulatability, i.e., A_2 is defined as the combination of A_1 and a simulator Sim that depends only on $(\hat{M}_1, S)$.

7 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 6. This is shown in Figure 5, together with the detailed ports of $\mathsf{Sim}_{\mathcal{H}}$ and a sketch of the messages to be handled. 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.

7.1 States of the Simulator

The simulator essentially contains the cryptographic objects that the adversary has seen or sent, together with the handles used for the corresponding terms in the Dolev-Yao-style model. This is represented by database D_{a} with the following attributes:

– $hnd_{\mathsf{a}} \in \mathcal{HNDS}$ 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.
– $word \in \{0,1\}^*$ is the real bitstring.

– 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^*).

A variable $curhnd_{\mathsf{a}}$ denotes the current size of D_{a}, except temporarily within an algorithm 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 $\mathsf{out_a}?$; cf. Section 4.1.

7.2 Simulating Sent Messages

When $\mathsf{Sim}_{\mathcal{H}}$ receives an "unsolicited" input from $\mathsf{TH}_{\mathcal{H}}$, this is the result of a send command by an honest user and thus of the form $m = (u, v, \mathsf{i}, l^{\mathsf{hnd}})$ for an insecure channel, and similar for other channel types. $\mathsf{Sim}_{\mathcal{H}}$ 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 $j \le \mathsf{max_len}(k)$, and $m_i^{\mathsf{hnd}} \le \mathsf{max_hnd}(k)$ if $m_i^{\mathsf{hnd}} \in \mathcal{HNDS}$ and otherwise $|m_i^{\mathsf{hnd}}| \le \mathsf{max_len}(k)$ (with certain domain expectations in the arguments m_i^{hnd} that are automatically fulfilled in interaction with $\mathsf{TH}_{\mathcal{H}}$, also for the extensions of the command adv_parse for the new symmetric-authentication 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 using $type$-specific subroutines.

We have to define the subroutines for Step 3 for symmetric authentication.

Definition 10 *(Producing simulated real messages for symmetric authentication) For the symmetric-authentication types, the algorithm* id2real *uses the following (stateful) subroutines in Step 4.*

– *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* $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, ())$.

7.3 Simulating Dolev-Yao-style Terms from Real Network Inputs

When $\mathsf{Sim}_{\mathcal{H}}$ receives an input 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} by a recursive algorithm $l^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(l)$ (with side-effects), and outputs $\mathsf{adv_send_i}(w, u, l^{\mathsf{hnd}})$ at port $\mathsf{in_a}!$. The algorithm real2id recursively parses the real message, builds up a corresponding term in $\mathsf{TH}_{\mathcal{H}}$, and enters all messages into D_{a}.

For an arbitrary message $m \in \{0,1\}^+$, $m^{\mathsf{hnd}} \leftarrow \mathsf{real2id}(m)$ works as follows. If there is already a handle m^{hnd} with $D_{\mathsf{a}}[m^{\mathsf{hnd}}].word = m$, it returns that. Else it sets $(type, arg) := \mathsf{parse}(m)$ and calls a type-specific algorithm $add_arg \leftarrow \mathsf{real2id_}type(m, arg)$. After this, real2id sets $m^{\mathsf{hnd}} := curhnd_{\mathsf{a}}$++ and $D_{\mathsf{a}} :\Leftarrow (m^{\mathsf{hnd}}, m, add_arg)$. We have to provide the type-specific algorithms for the new symmetric-authentication types.

Definition 11 *(Entering terms from real messages for symmetric authentication)*

- $add_arg \leftarrow \mathsf{real2id_ska}(m, ())$. *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* $D_{\mathsf{a}}[pka_{aut^{\mathsf{hnd}}}].add_arg[1] = \mathsf{honest}$. *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* $()$.

8 Security Proof

Our security claim is that the realization of the Dolev-Yao-style model extended with symmetric authentication is as secure as the Dolev-Yao-style model with symmetric authentication in the sense of Definition 9.

Let $RPar$ be the set of valid parameter tuples for the real system, consisting of the number $n \in \mathbb{N}$ of participants, secure signature, encryption, and symmetric authentication schemes $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{A}$, and length functions and bounds L'. For $(n, \mathcal{S}, \mathcal{E}, \mathcal{A}, L') \in RPar$, let $Sys^{\mathsf{cry_sym_auth,real}}_{n,\mathcal{S},\mathcal{E},\mathcal{A},L'}$ be the resulting real cryptographic library. Further, let the corresponding length functions and bounds of the ideal system be formalized by a function $L := \mathsf{R2Ipar}(\mathcal{S}, \mathcal{E}, \mathcal{A}, L')$, where the extension to the newly added length functions for symmetric authentication, i.e., $\mathsf{ska_len}^*$ and $\mathsf{aut_len}^*$, was given in Section 5.2. Moreover, we require that the function $\mathsf{max_in_a}(k)$ contained in L is a sufficiently large polynomial; we give a sufficient lower bound for $\mathsf{max_in_a}(k)$ in Section 8.3. Let $Sys^{\mathsf{cry_sym_auth,id}}_{n,L}$ be the ideal cryptographic library with parameters n and L.

Theorem 1 *(Security of the Dolev-Yao-style Model with Symmetric Authentication) For all parameters $(n, \mathcal{S}, \mathcal{E}, \mathcal{A}, L') \in RPar$, we have*

$$Sys^{\mathsf{cry_sym_auth,real}}_{n,\mathcal{S},\mathcal{E},\mathcal{A},L'} \geq Sys^{\mathsf{cry_sym_auth,id}}_{n,L},$$

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

Recall that we already defined a simulator in Section 7 in order to prove Theorem 1. 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, $u \in \mathcal{H}$ from the combination $\mathsf{THSim}_\mathcal{H}$ of $\mathsf{TH}_\mathcal{H}$ and $\mathsf{Sim}_\mathcal{H}$ (for all sets $\mathcal{H} \subseteq \{1, \ldots, n\}$ of indices indicating the correct machines). We do not repeat the precise definition of "combinations" here; it can be found in [33].

The proof is essentially a bisimulation. This means to define a mapping between the states of two systems (Section 8.2) and a sufficient set of invariants (Section 8.4) 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 the outputs are identically distributed (Section 8.5, 8.6, and 8.7). However, the states of our two 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 machine $\mathsf{C}_\mathcal{H}$, from which both $\mathsf{THSim}_\mathcal{H}$ and $\mathsf{M}_\mathcal{H}$ can be derived. The two derivations are two mappings, and we perform the two bisimulation proofs in parallel. By the transitivity of indistinguishability (of the families of views of the same A and H in all three configurations), we obtain the desired result. This is shown in Figure 6.

Specific aspects of this bisimulation, all as in [6], are the following. First, certain "error sets" of traces remain where the bisimulation fails. At the end, in Section 8.8, we show that the union of all error sets has negligible probability if the underlying primitives, here the symmetric authentication scheme, are secure;

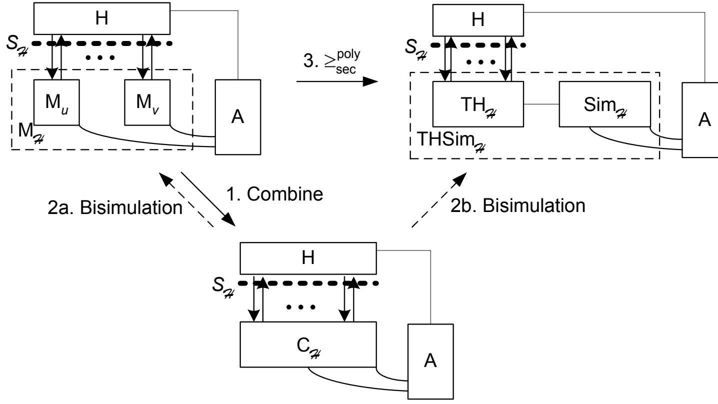


Fig. 6 Overview of the simulatability proof.

this is sufficient for computational indistinguishability. Secondly, we have a probabilistic invariant “strongly correct arguments”. Thirdly, 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 guess hitting an already existing truly random value is negligible, we can only exploit this if the adversary got no information (in the Shannon sense) about this value. We therefore have to show that the adversary did not even receive any *partial* information about this value. Partial information could be derivable since, e.g., the value was hidden within a nested term. We show the absence of partial information flow via the absence of static information flow, and the flow specification is the invariant “word secrecy”.

8.1 Combined Machine

The combined machine $\mathsf{C}_{\mathcal{H}}$ mainly contains a database D^* of all existing cryptographic objects. It is structured like D in $\mathsf{TH}_{\mathcal{H}}$, but with the following additional attributes:

- $word \in \{0,1\}^*$ contains a real bitstring as in $\mathsf{M}_{\mathcal{H}}$ or $\mathsf{Sim}_{\mathcal{H}}$ under the same handle(s).
- $parsed_u \in \{\mathsf{true}, \mathsf{false}\}$ for $u \in \mathcal{H}$ is $\downarrow$ if $hnd_u = \downarrow$ for the same entry; otherwise true indicates that the entry would be parsed in D_u, and false that it would still be of type null. As entries of type pka do not exist in the real system, we always have $parsed_u = \downarrow$ for them.
- $owner$ for secret keys and authenticators is adv if the key or the authenticator was first received from the adversary, otherwise 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 x whose first handle $x.hnd_u$ is for $u \in \mathcal{H}$ gets the word that M_u would contain under this handle, and otherwise that from $\mathsf{Sim}_{\mathcal{H}}$. Thus, essentially, entries created due to basic commands from H get the words that $\mathsf{M}_{\mathcal{H}}$ would construct, while words received in network inputs from A are parsed completely and entered as by $\mathsf{Sim}_{\mathcal{H}}$. Outputs to H are made as in $\mathsf{TH}_{\mathcal{H}}$, outputs to A as in $\mathsf{M}_{\mathcal{H}}$.

8.2 Derivations

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

- Let ω abbreviate word lookup, i.e., $\omega(i) := D^*[i].word$ if $i \in \mathcal{HNDS}$, else $\omega(i) := i$. Let ω^*, applied to a list, denote that ω is applied to each element.
- We give most derived variables and entire machine states a superscript *, because in the bisimulation we have to compare them with the "original" versions. We make an exception with some variables of $\mathsf{THSim}_{\mathcal{H}}$ that are equal by construction in $\mathsf{C}_{\mathcal{H}}$; in particular D^* is $\mathsf{C}_{\mathcal{H}}$'s extended database and the derived D-part for $\mathsf{TH}_{\mathcal{H}}$ is immediately called D again.
- Let $\mathsf{owners}(x) := \{u \in \mathcal{H} \cup \{\mathsf{a}\} \mid x.hnd_u \neq \downarrow\}$ denote the set of owners for $x \in D$. If $|\mathsf{owners}(x)| = 1$, we write $\mathsf{owner}(x)$ for the element of $\mathsf{owners}(x)$.

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.

Definition 12 *(Derivations from $\mathsf{C}_{\mathcal{H}}$ to $\mathsf{THSim}_{\mathcal{H}}$ and $\mathsf{M}_{\mathcal{H}}$) Given a state of $\mathsf{C}_{\mathcal{H}}$, the states of the individual machines consist of the following components.*

$\mathsf{TH}_{\mathcal{H}}$: *$D$: 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^*_{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 $\mathsf{C}_{\mathcal{H}}$.

8.3 Properties of the Ideal System and the Simulator

All properties shown about the ideal system in Lemmas 4.1 and 4.2 of [6] still hold after adding symmetric authentication, 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 $i \in \mathbb{N}_0$.

The simulator is polynomial-time.

Further, no handle output by $\mathsf{TH}_\mathcal{H}$ is rejected by $\mathsf{Sim}_\mathcal{H}$, and the counters $steps_{\mathsf{out_a}?}$ of $\mathsf{Sim}_\mathcal{H}$ and $steps_{\mathsf{in_a}?}$ of $\mathsf{TH}_\mathcal{H}$ never reach their bounds in $\mathsf{THSim}_\mathcal{H}$. This is shown as in [6], except for the new bound $\mathsf{max_in_a}(k)$ for $steps_{\mathsf{in_a}?}$ and $steps_{\mathsf{out_a}?}$. The function $\mathsf{max_in_a}(k)$ not only has to ensure polynomial runtime but also has to be large enough to ensure correct functional behavior by never being reached in a simulation. In [6], an upper bound on the inputs needed in the simulation is derived from the security proof and used as an instantiation of $\mathsf{max_in_a}(k)$, but any larger polynomial would have been sufficient. In this work we adapt this upper bound as follows. Because of the interaction of $\mathsf{TH}_\mathcal{H}$ and $\mathsf{Sim}_\mathcal{H}$ in real2id, the number of these steps increases 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. However, only a polynomial number of authenticators and keys can be created (a coarse bound is $n \cdot \mathsf{max_in}(k)$ for entries of the honest users plus the polynomial runtime of A for the remaining ones), and thus $\mathsf{max_in_a}(k)$ remains polynomial as required in [6].

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

For the bisimulation, we need invariants about $\mathsf{C}_\mathcal{H}$. In the original proof, 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 by symmetric authentication. In the following, we present the important invariants for the new proof. Each of them already occurred in [6], except for "correct verification", which is trivially invariant for the original inputs since it only makes statements about our new types. Each existing invariant is generalized for dealing with our new types, without new conditions on database entries of old types.

- *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 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 A and 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 known to the adversary, i.e., if $D^*[i].hnd_\mathsf{a} \neq \downarrow$ and $D^*[i].type \in \{\mathsf{pks}, \mathsf{pks}\}$, then also $D^*[i+1].word$.[2]
- symmetric authentication keys for which a corresponding authenticator is known to the adversary, i.e., all words $D^*[i].word$ with $D^*[i].type = \mathsf{ska}$ for which there exists an entry $D^*[j]$ with $D^*[j].hnd_\mathsf{a} \neq \downarrow$, $D^*[j].type = \mathsf{aut}$, and $i-1 \in D^*[j].arg$.

"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.

- *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:[3]
  - If $type = \mathsf{aut}$, then arg^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 $(\mathsf{aut}, sr, r, l, aut) := D^*[i].word$, $(l^\mathsf{ind}, pka_1^\mathsf{ind}, \ldots, pka_j^\mathsf{ind})) :=$

[2] 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. This holds similarly for symmetric authentication keys which are captured in the next bullet.

[3] 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.

$D^*[i].arg$, 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 $\mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true}$.

8.5 Bisimulation of Basic Commands

Recall that we have to show that except for certain error sets, every external input to the two systems in mapped states fulfilling the invariants keeps the systems in mapped states fulfilling the invariants, and that the outputs are identically distributed. We first consider the effects of a basic command c input at a port $\mathsf{in}_u?$ with $u \in \mathcal{H}$. Recall that the actions of $\mathsf{C}_\mathcal{H}$ on a large part of its state are by definition equal to those of $\mathsf{TH}_\mathcal{H}$, and so is $\mathsf{C}_\mathcal{H}$'s output at $\mathsf{out}_u!$. We will not always mention this again. 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 M_u set $ska^{\mathsf{hnd}} := curhnd_u{+}{+}$, and make two entries in case of $\mathsf{TH}_\mathcal{H}$, respectively one entry in case of M_u. In $\mathsf{C}_\mathcal{H}$ this gives $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{pka}, arg := (), len := 0)$ and $D^* :\Leftarrow (ind := size{+}{+}, type := \mathsf{ska}, arg := (ind - 1), hnd_u := ska^{\mathsf{hnd}}, len := \mathsf{ska_len}^*(k), parsed_u := \mathsf{true}, word := sk^*)$ 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 the definition of $\mathsf{make_auth_key}$ implies $\mathsf{ska_len}^*(k) = \mathsf{list_len}(|\mathsf{ska}|, \mathsf{ska_len}(k), \mathsf{nonce_len}(k)) = |sk^*|$; nothing is required for type pka. Under "correct arguments", nothing is required for type ska and 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 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 $aut^{\mathsf{hnd}} := curhnd_u{+}{+}$ and makes a new entry $D :\Leftarrow (ind := size{+}{+}, type := \mathsf{aut}, arg := (l^{\mathsf{ind}}, ska^{\mathsf{ind}} - 1), hnd_u := aut^{\mathsf{hnd}}, len := length)$.
  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 M_u sets $aut^{\mathsf{hnd}} := curhnd_u{+}{+}$ 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 make_auth with the predicate in 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 M_u return $\downarrow$ if $D^*[aut^{\mathsf{ind}}].type \neq \mathsf{aut}$ or if $D^*[ska^{\mathsf{ind}}].type \neq \mathsf{ska}$ (indeed 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 false if $pka_1^{\mathsf{ind}} = \downarrow$ or $ska^{\mathsf{ind}} - 1 \notin \{pka_1^{\mathsf{ind}}, \ldots, pka_j^{\mathsf{ind}}\}$, and true otherwise. M_u outputs 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 M_u return $\downarrow$ if $D^*[aut^{\mathsf{ind}}].type \neq \mathsf{aut}$. (Indeed 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 $l^{\mathsf{hnd}} := curhnd_u{+}{+}$. By "word uniqueness" and "correct derivation", M_u does not find another entry with the word l, and thus makes a new entry $(l^{\mathsf{hnd}}, l, \mathsf{null}, ())$ 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.

8.6 Bisimulation of Send Commands from Honest Users

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^{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 M_u always outputs $l := D^*[l^{\mathsf{ind}}].word$. An inductive proof is used that id2real retains all invariants and produces the right outputs. By inspection of 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, 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 [6], 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 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. This 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.

8.6.1 Authentication Keys If $type$ = 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_{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 $type$ = ska, Let $pka^{\mathsf{hnd}} := m_1^{\mathsf{hnd}}$. By construction, we have $D^*[pka^{\mathsf{hnd}}].type = \mathsf{pka}$.[4] 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.

8.6.2 Authenticators If $type$ = aut, "strongly correct arguments" implies that arg^{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

[4] 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.

$D_a[pka_1^{ind}].add_arg[1] = \mathsf{honest}$, then "correct derivation" of D_a implies $D_a[pka_1^{hnd}].add_arg = (\mathsf{honest}, sk^*)$, where $pka_1^{hnd} = D^*[pka_1{}^{ind}]$. If $D_a[pka_1^{ind}].add_arg = (\mathsf{adv})$ then "correct derivation" of D_a implies $D_a[pka_1^{ind} + 1].word = sk^*$. 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.

8.7 Bisimulation of Network Inputs 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 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 $l_a^{hnd} \leftarrow \mathsf{real2id}(l)$ to parse the input. Using a lemma from [6], we only have to show the following properties of each call $l_a^{hnd} \leftarrow \mathsf{real2id}(l)$ with $0 < |l| \leq \mathsf{max_len}(k)$ and $l \in Pub_Var$:

- At the end, $D^*[hnd_a = l_a^{hnd}].word = l$ and $D^*[hnd_a = l_a^{hnd}].type \notin \{\mathsf{sks}, \mathsf{ske}\}$.
- "Correct derivation" of D_a and $curhnd_a$.
- 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 $D^*[hnd_a = l_a^{hnd}]$ 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^{hnd} with $D_a[m^{hnd}].word = m$, real2id returns that. The postulated output condition is fulfilled by "correct derivation", and the others because no state changes are made. Otherwise, the word m is not yet present in D_a. Then id2real sets $(type, arg) := \mathsf{parse}(m)$. This yields $type \in typeset \setminus \{\mathsf{sks}, \mathsf{ske}\}$. As parse is a functional algorithm, no invariants are affected. Then id2real calls an algorithm $add_arg \leftarrow \mathsf{real2id_}type(m, arg)$ with side-effects.

Finally it sets $m^{hnd} := curhnd_a{++}$ and $D_a :\Leftarrow (m^{hnd}, m, add_arg)$.

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

8.7.1 Authentication Keys

The algorithm $\mathsf{real2id_ska}(m, ())$ calls $ska^{hnd} \leftarrow \mathsf{gen_auth_key}()$ at $\mathsf{in_a}!$ and sets $D_a :\Leftarrow (curhnd_a{++}, \epsilon, (\mathsf{adv}))$ for the key identifier and $add_arg := ()$ for the secret key.

Recall that the upcoming loop over all aut^{hnd} can only modify the database D^* by outputting a command adv_fix_aut_validity, which does not create new entries.

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

In $\mathsf{C}_\mathcal{H}$, the key identifier entry results in $D^* :\Leftarrow (ind := size{++}, type := \mathsf{pka}, arg := (), hnd_a := pka^{hnd}, len := 0)$. 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_a, because m is not present in D_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_a. "Correct arguments" and "word secrecy" are obvious. For "correct length", nothing is required for type 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^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 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 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_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 $D^*[aut^\mathsf{hnd}].word \notin Pub_Var$.

8.7.2 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 parse. Hence when 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 $Ska := \{ska^{\mathsf{hnd}} \mid D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].type = \mathsf{ska} \wedge D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].word[3] = sr \wedge \mathsf{atest}_{sk}(aut, (r, l)) = \mathsf{true} \text{ for } sk := D^*[hnd_{\mathsf{a}} = ska^{\mathsf{hnd}}].word[2]\}$.

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 make_auth_key. This means that if two such distinct elements existed, the nonces sr collided in two executions of 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^{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_{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^{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_{a}. This means that the adversary has guessed an authenticator that existed in $\mathsf{TH}_{\mathcal{H}}$ but that he has not sent 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 $x \notin Pub_Var$.

After that, $\mathsf{Sim}_{\mathcal{H}}$ calls $\mathsf{adv_fix_aut_validity}(ska^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ for every $ska^{\mathsf{hnd}} \in Ska$, 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^{hnd} for j will be contained in 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 $i \notin Pub_Var$ because of $i.hnd_{\mathsf{a}} = \downarrow$.

Case 2: A Valid Key Exists in D_{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 $Ska \neq \emptyset$. $\mathsf{Sim}_{\mathcal{H}}$ then picks $ska^{\mathsf{hnd}} \in Ska$ 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 ska and to parsing m in case of 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 $D :\Leftarrow (ind := size\text{++}, type := \mathsf{aut}, arg := (l, ska - 1), hnd_{\mathsf{a}} := aut^{\mathsf{hnd}}, len := length, word = m)$. "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_{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 $x \notin Pub_Var$.

After that, $\mathsf{Sim}_{\mathcal{H}}$ calls $v \leftarrow \mathsf{adv_fix_aut_validity}(ska'^{\mathsf{hnd}}, l^{\mathsf{hnd}})$ for every $ska'^{\mathsf{hnd}} \in Ska \setminus \{ska^{\mathsf{hnd}}\}$, 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'^{hnd} for j will be contained in 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 $i \notin Pub_Var$ 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 gen_auth_key, and thus as $sk \leftarrow \mathsf{gen_A}(1^k)$.

Case 3: No Valid Key Exists in D_{a}. Now assume that $Ska = \emptyset$. This either means that no key in D_{a} has a suitable nonce sr or that the authenticator test fails for all keys in D_{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_{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 gen_auth_key, and thus as $sk \leftarrow \mathsf{gen_A}(1^k)$.

8.8 Error Sets

We finally have to show that the union of all error sets has only negligible probability if the underlying cryptographic primitives are secure, i.e., the symmetric authentication scheme and the nonces used for tagging.

In the bisimulation, three error sets $Nonce_Coll$, $Nonce_Guess$, and $Auth_Forge$ were defined. They contain runs where two nonces collided, where the adversary has guessed a nonce value that he ideally has not yet seen, and where the adversary successfully forged an authenticator, respectively. Intuitively, such events should indeed only occur with negligible probability.

More precisely, we have three sequences of error sets, each indexed with the security parameter k, such as $(Auth_Forge_k)_{k \in \mathbb{N}}$. If each sequence has negligible probability, then so has the sequence of the set unions. Hence we now assume for contradiction that one sequence has a larger probability for certain polynomial-time users H and adversary A.

Recall that the elements of the error sets are runs of the combined machines $\mathsf{C}_{\mathcal{H}}$. The proofs rely on the fact that the execution of $\mathsf{C}_{\mathcal{H}}$ with H and A is polynomial-time. This has already been shown for the original model, and this also holds for 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 8.3.

8.8.1 Nonce Collisions The error set $Nonce_Coll$ occurs in Sections 8.5 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 $sr \leftarrow_R \{0,1\}^{\mathsf{nonce_len}(k)}$ (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.

8.8.2 Nonce Guessing The error set $Nonce_Guess$ occurs in Section 8.7.1 and 8.7.2. A run is put into this set if the adversary has guessed an existing nonce value that ideally he should not have seen. In all these cases we showed that the adversary had guessed the word of an entry $x \in D^*$ with $x.hnd_{\mathsf{a}} = \downarrow$, $x.word \not\in Pub_Var$, and $x.type \in \{\mathsf{ska}, \mathsf{aut}\}$. "Strongly correct arguments" implies that each of them contains a nonce part generated as $sr \leftarrow_R \{0,1\}^{\mathsf{nonce_len}(k)}$ for type ska and $r \leftarrow_R \{0,1\}^{\mathsf{nonce_len}(k)}$ for type 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 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.

8.8.3 Authenticator Forgery The error set $Auth_Forge$ occurs in Section 8.7.2 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 $sk \leftarrow \mathsf{gen}_{\mathsf{A}}(1^k)$.

In the combined machine $\mathsf{C}_{\mathcal{H}}$, this secret key sk was a component $D^*[sk^{\mathsf{ind}}].word[2]$ with $a \not\in \mathsf{owners}(D^*[sk^{\mathsf{ind}}])$. Thus it is only used if the command 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 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 $\mathsf{A}_{\mathsf{aut}}$ against the signer machine Aut from Definition 4 by letting $\mathsf{A}_{\mathsf{Aut}}$ execute $\mathsf{C}_{\mathcal{H}}$, using the given A and H as blackboxes. It only has to choose an index $i \leftarrow_R \{1, \ldots, n \cdot \mathsf{max_in}(k)\}$ 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 Aut instead. Hence the success probability of $\mathsf{A}_{\mathsf{aut}}$ 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.

9 Conclusion

We have shown how symmetric authentication can be treated in Dolev-Yao-style symbolic protocol proofs. Our abstraction is faithful, i.e., essentially it guarantees that a protocol proved over the abstraction can be realized safely with a well-defined cryptographic realization. The abstraction is not stand-alone – protocols using nested Dolev-Yao-style terms with symmetric authentication as the only cryptographic primitive would be quite exotic. Instead, we defined it as an add-on to an already proven Dolev-Yao-style model containing public-key encryption and signatures.

Many variants of the abstraction are conceivable. Deterministic instead of probabilistic authentication should be easy, while adding schemes with memory seems more complex with respect to specific adversary capabilities. Another task is to consider special cryptographic authentication schemes that realize the abstraction with fewer adversary capabilities, e.g., no authenticator transformation. One question is whether such schemes can be constructed efficiently from every more general scheme, another question is whether these adversary capabilities really hurt in any interesting protocol. Omitting the ability to retrieve the message from an authenticator seems less interesting since schemes with this property can be constructed from others simply by defining the pair (m, aut) of a message and its original authenticator as the new authenticator, and in most protocols this does not decrease efficiency.

The major novelty compared with the existing first proof of a Dolev-Yao-style abstraction under active attacks for public-key primitives was to treat the exchange of secret keys, in particular after these keys are first used. Then either the simulator has already simulated authenticators from honest participants to the adversary and later has to provide a suitable key – this proved no great problem given an abstraction that allows message retrieval. Or the adversary has first sent the authenticator, and the simulator later has to adapt its ideal representation to the new key knowledge. This we solved by allowing special authenticator terms with zero or more keys. On the whole, the new aspects, even though needed specifically to achieve reactive simulatability, influenced the Dolev-Yao-style abstraction of this symmetric primitive more than the overall proof technique. This is a good sign that new primitives can be added to the Dolev-Yao-style model in a modular way. As an outlook, however, let us mention that symmetric encryption needs more significant additions [5].

Acknowledgments

Comments from many people on this paper and the underlying ones were very helpful to (hopefully) make this presentation much clearer. In particular, we thank Ran Canetti, Anupam Datta, Ante Derek, Joshua Guttman, Ralf Küsters, Peeter

Laud, Daniele Micciancio, John Mitchell, Dusko Pavlovic, Andre Scedrov, Bogdan Warinschi, and Thomas Wilke.

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 and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 1–12, 2003. Full version in IACR Cryptology ePrint Archive 2003/121, Jun. 2003, `http://eprint.iacr.org/`.
5. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, 2004. Full version in IACR Cryptology ePrint Archive 2004/059, Feb. 2004, `http://eprint.iacr.org/`.
6. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003. Full version in IACR Cryptology ePrint Archive 2003/015, Jan. 2003, `http://eprint.iacr.org/`.
7. D. Beaver. Secure multiparty protocols and zero knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.
8. M. Bellare, 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.
9. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 3(1):143–202, 2000.
10. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001. Extended version in Cryptology ePrint Archive, Report 2000/67, `http://eprint.iacr.org/`.
11. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
12. S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 34–39, 1983.
13. S. Even, O. Goldreich, and A. Shamir. On the security of ping-pong protocols when implemented using the RSA (extended abstract). In *Advances in Cryptology: CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 1986.
14. O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

15. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – or – a completeness theorem for protocols with honest majority. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.
16. S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology: CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.
17. 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.
18. J. Herzog, M. Liskov, and S. Micali. Plaintext awareness via key registration. In *Advances in Cryptology: CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 548–564. Springer, 2003.
19. R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.
20. H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Crptology: CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 129–139. Springer, 1994.
21. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming (ESOP)*, pages 77–91, 2001.
22. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. 25th IEEE Symposium on Security & Privacy*, pages 71–85, 2004.
23. 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.
24. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 18–30, 1997.
25. C. Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.
26. M. Merritt. *Cryptographic Protocols*. PhD thesis, Georgia Institute of Technology, 1983.
27. S. Micali and P. Rogaway. Secure computation. In *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 392–404. Springer, 1991.
28. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.
29. J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.
30. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
31. 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`.
32. 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. Extended version (with Matthias Schunter) IBM Research Report RZ 3206, May 2000, `http://www.semper.org/sirene/publ/PfSWl_00ReactSimulIBM.ps.gz`.
33. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security &*

Privacy, pages 184–200, 2001. Extended version of the model (with Michael Backes) IACR Cryptology ePrint Archive 2004/082, `http://eprint.iacr.org/`.

34. P. Rogaway. Bucket hashing and its application to fast message authentication. In *Advances in Crptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 19954.
35. 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.
36. S. Schneider. Security properties and CSP. In *Proc. 17th IEEE Symposium on Security & Privacy*, pages 174–187, 1996.
37. A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.