A Simulation-Based Treatment of Authenticated Message Exchange [⋆]

Klaas Ole Kürtz, Henning Schnoor, and and Thomas Wilke

Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany
{kuertz|schnoor|wilke}@ti.informatik.uni-kiel.de

Abstract. Simulation-based security notions for cryptographic protocols are regarded as highly desirable, primarily because they admit strong composability and, consequently, a modular design. In this paper, we give a simulation-based security definition for two-round authenticated message exchange and show that a concrete protocol, 2AMEX-1, satisfies our security property, that is, we provide an ideal functionality for two-round authenticated message exchange and show that 2AMEX-1 realizes it securely. To model the involved public-key infrastructure adequately, we use a joint-state approach.

1 Introduction

Simulation-based security definitions for cryptographic protocols, see, for instance, [Can01,PW01,BPW04,Küs06], are attracting much attention, the reasons being that such security definitions "guarantee security even when a secure protocol [. . .] is used as a component of an arbitrary system" [Can01] and that they enable "modular proofs of security" [PW01].

As a consequence, a variety of cryptographic primitives such as asymmetric encryption and digital signatures have been treated following the simulation-based approach. There are, however, only few complex cryptographic protocols that have been tackled within the simulation-based framework. We are aware of [CK02,MN06,BCJ$^+$06,BP06,GMP$^+$08], where, for instance, Kerberos and the Yahalom protocol are treated.

In this paper, we deal with two-round authenticated message exchange protocols following the simulation-based approach. We (i) provide an ideal functionality for two-round authenticated message exchange protocols, $\mathcal{F}_{\mathrm{2AM}}$, (ii) provide an implementation, $\mathcal{P}_{\mathrm{2AMEX-1}}$, corresponding to a particular such protocol, 2AMEX-1, and (iii) prove the implementation of 2AMEX-1 to be secure, that is, prove that $\mathcal{P}_{\mathrm{2AMEX-1}}$ securely realizes the ideal functionality, in symbols $\mathcal{P}_{\mathrm{2AMEX-1}} \leq^{\mathrm{BB}} \mathcal{F}_{\mathrm{2AM}}$. (The superscript stands for black-box simulatability.)

The protocol 2AMEX-1, see [KSW09], which is a generic protocol for message authentication in a web service setting, is complex in several respects: it distinguishes between short-lived clients and long-lived servers; it uses digital signatures and therefore makes use of a public-key infrastructure; it requires

[⋆] This work was partially supported by the DFG under grant KU 1434/4-2.

only bounded memory; it uses nonces and timestamps to counter replay attacks; each client and each server has its own local clock. In [KSW09], 2AMEX-1 was proved to be secure in the Bellare-Rogaway framework as presented in [BR93].

Several simulation-based approaches have been developed over the last decade (see above). We could have used any of these approaches, but we have adopted the one by Küsters, see [Küs06], because it provides a very flexible addressing mechanism and easy-to-use joint-state theorems, see [KT08a]. The latter is especially useful in the analysis of 2AMEX-1, because it allows us to show with only little effort that 2AMEX-1 works securely with a simple, but realistic public-key infrastructure. Although Küsters' setting comes in handy in many respects, it also has some shortcomings, which become evident from our analysis and are discussed in this paper.

We start with the sketch of Küsters' model in Section 2, go on with a description of the setting and the ideal functionalities in Section 3 and a description of the implementation for 2AMEX-1 in Section 4, and conclude with our main result and a discussion in Sections 5 and 6.

We are grateful to Max Tuengerthal for helpful comments.

2 Simulation-Based Security

In this section, we give a high-level description of the simulation-based framework from [Küs06], which is referred to as the *IITM framework*, where IITM stands for *inexhaustible interactive Turing machine*.

In the IITM framework cryptographic protocols and the environment they are run in (including the adversary) are modeled as concurrent, polynomial-time, probabilistic, interactive, replicable Turing machines. Here, "concurrent" refers to an interleaving semantics, that is, only one IITM is active at a time and there is a mechanism that determines which IITM is activated next; "replicable" refers to a mechanism which allows certain machines, the so-called banged machines, to be instantiated several times (and run concurrently); "interactive" means that the machines can communicate by sharing tapes, more precisely: an output tape of one machine can be the input tape of another machine. From a security point of view, it is important that systems of IITM's can be simulated in polynomial time. To achieve this, it is, however, not enough to require that the individual IITM's are polynomial-time, because two IITM's "playing ping pong" could double their outputs on each activation, leading to an overall exponential running time. For that reason the IITM framework imposes certain restrictions on how machines are interconnected, based on a partition of tapes into consuming and enriching. Roughly speaking, the overall length of the output of one IITM up to a certain point may be polynomial in the overall length of the input on enriching tapes up to the same point, but there must not be any cycle of enriching tapes. This is less restrictive than requiring that each IITM runs in time polynomial in the security parameter; it allows to process inputs of arbitrary size.

To illustrate the IITM framework consider Figure 1 and first focus on the box labeled $\mathcal{F}_{2\mathrm{AM}}$. This box represents a model of two-round authenticated message

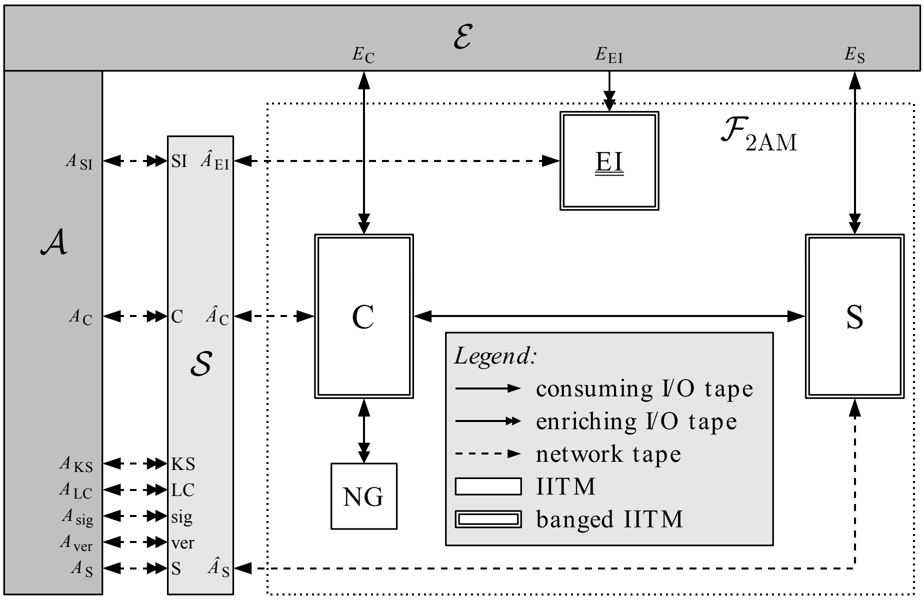

Fig. 1. Ideal functionality for two-round message authentication

exchange protocols (details follow in the next section); it contains four machines which represent an actual protocol: C, S, EI, and NG, of which the first three are banged (can be replicated), and the last one is not. Every instance of machine C is connected with machine NG, in both directions. The corresponding input tape of NG is enriching, while the input tape of C is not.

There are two types of connections crossing the borders of $\mathcal{F}_{2\text{AM}}$: solid connections representing tapes classified as I/O tapes and dashed connections representing tapes classified as network tapes. I/O tapes should roughly be thought of as tapes communicating with "users" of the system, whereas network tapes are tapes where the adversary can interfere.

In Figure 1, the adversary, represented by an IITM denoted $\mathcal{A}$, is not connected directly with $\mathcal{F}_{2\text{AM}}$. Rather, there is a mediator between $\mathcal{A}$ and $\mathcal{F}_{2\text{AM}}$, namely an IITM $\mathcal{S}$ called simulator. The situation is typical for simulation-based security: a simulator "translates" network traffic to make a system (in this case $\mathcal{F}_{2\text{AM}}$) seem equivalent to another one (usually a "real" system $\mathcal{P}$, see below) to an outside observer consisting of an environment machine $\mathcal{E}$ (taking over the role of all users) and an adversary $\mathcal{A}$.

Another feature of Figure 1 not discussed yet has to do with how different instances of the same machine are addressed. Underlining the name of a machine indicates the usage of a generic addressing mechanism provided by the IITM framework, which works by using prefixes of messages as identifiers for instances. In Figure 1 the machine EI is underlined twice, which adds two prefixes for addressing, that is, a hierarchical addressing mechanism is used. We use it to model multi-user multi-session instances.

The formal way to specify the system represented by the box $\mathcal{F}_{\mathrm{2AM}}$ in Figure 1 is by the expression

$$\mathcal{F}_{\mathrm{2AM}} = !\mathcal{F}_{\mathrm{C}} \mid !\mathcal{F}_{\mathrm{S}} \mid \mathcal{F}_{\mathrm{NG}} \mid !\underline{\underline{\mathcal{F}_{\mathrm{EI}}}} \ , \tag{1}$$

where $\mathcal{F}_{\mathrm{C}}$, $\mathcal{F}_{\mathrm{S}}$, $\mathcal{F}_{\mathrm{NG}}$, and $\mathcal{F}_{\mathrm{EI}}$ denote (descriptions of) the underlying IITM's, and $\mid$ denotes an operator for composing machines.

In the IITM framework, security of a protocol is defined as follows. First, one describes a system of IITM's, $\mathcal{F}$, which works in an ideal fashion in every setting where an environment and an adversary are connected to it, that is, how one would expect a perfect protocol to work. Such a system is called an ideal functionality. Then, given a real protocol, one describes a system of IITM's, $\mathcal{P}$, which works just the way the real protocol would work in every setting where an environment and an adversary are connected to it. Now, $\mathcal{P}$ is considered secure if there is a simulator IITM $\mathcal{S}$ with the following property. For every environment machine $\mathcal{E}$ and every adversary machine $\mathcal{A}$, the system composed of $\mathcal{P}$, $\mathcal{E}$, and $\mathcal{A}$ is computationally indistinguishable from the system composed of $\mathcal{F}$, $\mathcal{E}$, $\mathcal{A}$, and $\mathcal{S}$. As explained above communication between these machines is restricted as follows: all external network connections of $\mathcal{F}$ are handled by the simulator $\mathcal{S}$; the adversary may only communicate with $\mathcal{F}$ using the network interface provided by the simulator; and the environment may only communicate with $\mathcal{F}$ using I/O connections. Hence, the system composed of $\mathcal{F}$ and the simulator (translating network messages) is "equivalent" to $\mathcal{P}$. In other words, every attack on the real protocol can be transferred into the ideal system.

If the above condition is satisfied, then $\mathcal{P}$ securely realizes (or implements) $\mathcal{F}$, denoted by $\mathcal{P} \leq^{\mathrm{BB}} \mathcal{F}$ (for *black-box simulation*).

3 Two-Round Authenticated Message Exchange

We start with a description of the general scenario. In a session of a two-round authenticated message exchange protocol (2AM protocol) a client sends a request to a server and expects to receive an appropriate response. This is, for instance, the case for web service calls, see, e. g., [ML07,LB07] and remote procedure calls, see, e. g. [Sun98,Win99]. Observe that for these protocols to make sense the request and response messages include payloads.

In a 2AM protocol the request and the response messages are required to be secured in such a way that (i) both client and server can verify that the messages they receive are authentic, (ii) the server accepts no message twice (payloads, on the contrary, may be received twice, but only in different messages), and (iii) if the client receives a response, it can be sure which of his requests the response refers to. Note that the same client may have multiple sessions with the same or different servers in parallel, but each session has only two rounds.

```
Tapes: C $\twoheadleftarrow\!\!\longrightarrow E_\mathrm{C}$, C $\dashleftarrow\!\dashrightarrow \hat{A}_\mathrm{C}$, C $\longleftrightarrow$ S, C $\longleftarrow\!\!\twoheadrightarrow$ NG
Initialization: $c = s = r = \varepsilon$, $n = 0$, $state = \mathsf{Init}$, $cor = \mathsf{false}$
Steps: loop
    Send a request to the server:
    if $(c', (\mathsf{Client}, s'), \mathsf{Init})$ received from $E_\mathrm{C}$
        Let $state = \mathsf{OK}$, $c = c'$ and $s = s'$.
        Send $(c, (\mathsf{Client}, s), \mathsf{GetNonce})$ to NG.
        Recv $(c, (\mathsf{Client}, s), \mathsf{Nonce}, r')$ from NG, let $r = r'$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Nonce}, r)$ to $E_\mathrm{C}$.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Request}, p_c, 1^{n'})$ from $E_\mathrm{C}$, let $n = n'$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Request}, p_c, n)$ to $\hat{A}_\mathrm{C}$.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Request}, \mathsf{Send})$ from $\hat{A}_\mathrm{C}$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Request}, p_c)$ to S.
    Receive and process a response from the server:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p_s)$ received from S
        If $state \neq \mathsf{OK}$ or $|p_s| > n$, abort.
        Let $state = \mathsf{Stopped}$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Response}, p_s)$ to $E_\mathrm{C}$.
Corruption: $\mathtt{Corr}(cor, \mathsf{true}, state \neq \mathsf{Init}, \varepsilon, \hat{A}_\mathrm{C}, \{E_\mathrm{C}\}, E_\mathrm{C})$
CheckAddress: Accept the initialization message only once. Check for $c$, $s$, and $r$ as soon as each
    one has been set.
```

Fig. 2. The client functionality $\mathcal{F}_\mathrm{C}$

3.1 Overview of the Ideal Functionality

Our model of the ideal functionality for 2AM protocols consists of four functionalities, see Figure 1: a client $\mathcal{F}_\mathrm{C}$ (defined in Figure 2), a server $\mathcal{F}_\mathrm{S}$ (defined in Figure 3), a nonce generator $\mathcal{F}_\mathrm{NG}$ (defined in Appendix B.4), and an enriching input functionality $\mathcal{F}_\mathrm{EI}$ (defined in Appendix B.5). The ideal functionality $\mathcal{F}_\mathrm{2AM}$ is the composition of these functionalities, as defined in (1).

One instance of the client functionality handles exactly one session between a client identity and a server, i. e., after initialization it basically (i) receives a request from the environment and encapsulates it in a message to a server, and (ii) receives a response from the server and forwards its contents to the environment. One instance of the server functionality also handles exactly one session; as with the client, it consists of receiving a request and sending a response. The nonce generator generates globally unique session identifiers (numbers used once, nonces) to distinguish multiple sessions between two parties. The enriching input functionality passes bits from an enriching input tape to the adversary. These bits are necessary to give the adversary additional capabilities as explained in Section 4.3.

3.2 Ideal Client Functionality

When the environment wants to start a new session, it provides the client with the identity of a server the client is supposed to communicate with. The client then responds with a nonce, which can be viewed as a handle, i. e., it allows the environment to distinguish different sessions this client is involved in.

The environment can now pass the payload of the request message to the client as well as enough resources to process a possible response from the server. The client then notifies the adversary that a message is ready to be sent. If the

```
Tapes: S $\twoheadleftarrow\!\!\longrightarrow E_{\mathrm{S}}$, S $\dashleftarrow\!\dashrightarrow \hat{A}_{\mathrm{S}}$, S $\longleftrightarrow$ C
Initialization: $s = c = r = p_s = \varepsilon$, $n = 0$, $state = \mathsf{Init}_0$, $cor = \mathsf{false}$
Steps: loop
    Initialization by the environment:
    if $(s', (\mathsf{Server}), \mathsf{Init}, 1^{n'})$ received from $E_{\mathrm{S}}$
        If $state \neq \mathsf{Init}_0$, abort. Let $s = s'$ and $n = n'$.
        Send $(s, (\mathsf{Server}), \mathsf{Init}, n)$ to $\hat{A}_{\mathrm{S}}$.
        Recv $(s, (\mathsf{Server}), \mathsf{Init}, \mathsf{OK})$ from $\hat{A}_{\mathrm{S}}$.
        Let $state = \mathsf{Init}_1$.
    Receive and process a request from the client:
    if $(c', (\mathsf{Client}, s, r'), \mathsf{Request}, p_c)$ received from C
        If $state \neq \mathsf{Init}_1$ or $|p_c| > n$, abort. Let $state = \mathsf{OK}$, $c = c'$, and $r = r'$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Request}, p_c)$ to $E_{\mathrm{S}}$.
    Receive a response payload from the environment:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p)$ received from $E_{\mathrm{S}}$
        Let $p_s = p$. Send $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p_s)$ to $\hat{A}_{\mathrm{S}}$.
    Deliver a response to the client:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, \mathsf{Send})$ received from $\hat{A}_{\mathrm{S}}$ and not $cor$
        If $state \neq \mathsf{OK}$, abort. Let $state = \mathsf{Stopped}$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p_s)$ to C.
    Send an error message to the environment:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, \mathsf{Error})$ received from $\hat{A}_{\mathrm{S}}$
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Response}, \mathsf{Error})$ to $E_{\mathrm{S}}$.
Corruption: $\mathtt{Corr}(cor, \mathsf{true}, state \neq \mathsf{Init}_0, \varepsilon, \hat{A}_{\mathrm{S}}, \{E_{\mathrm{S}}\}, E_{\mathrm{S}}, s)$
CheckAddress: Accept the initialization message only once. Check for $s$, $c$, and $r$ as soon as each
    one has been set.
```

Fig. 3. The server functionality $\mathcal{F}_{\mathrm{S}}$

adversary (ever) allows the transfer, the message is written to the incoming tape of the server. This models the adversary's ability to delay or drop messages on the network.

When the server transfers a response (which is not too large), the client simply unwraps it and forwards the contents to the environment. The details are spelled out in Figure 2.

A special mode of computation of IITM's, CheckAddress, is used in the last line of IITM definitions like Figure 2 to determine whether an incoming message is addressed to the current instance of the client IITM. If a message is rejected by all running instances, a new instance of the client IITM is started since the client IITM is banged in $\mathcal{F}_{2\mathrm{AM}}$. In addition, we use the corruption macro `Corr` from [KT08a] (with a slightly extended addressing mechanism) to allow a uniform treatment of corruption of clients and servers in both the ideal and the real world, see Appendix B.3.

3.3 Ideal Server Functionality

To start a session on the server side, the environment sends a message to the server with the identity it is supposed to receive messages for and the maximal length of an incoming request message.

Upon receiving a request from a client, the server unwraps it and forwards the request payload to the environment. Now the environment can respond by passing a response payload to the server functionality. The server asks the adversary, who has three options: It can either approve the sending of the payload, in which case the server delivers the message directly to the client. Secondly, the adversary can ignore the response, in which case the server sends no message at all. Thirdly, the adversary can also explicitly deny processing the payload, which results in an error message being sent to the environment.

The first two options again model that the adversary may intercept and delay network traffic. The third type of reaction models that in our implementation the server may reject messages due to bounded memory and notify the environment of the rejection.

4 Implementation of the 2AMEX-1 Protocol

In this section, we describe a system of IITM's implementing the 2AMEX-1 protocol, which is a 2AM protocol in the above sense and described in detail in [KSW09]. First, we give an informal introduction into the protocol.

4.1 The Protocol 2AMEX-1

In 2AMEX-1, an authenticated message exchange between a client with identity c and a server with identity s works roughly as follows.

1. a) c is asked by the environment to send the request p_c
   b) c sends $\{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c}$ to s
   c) s checks whether the message is admissible and if not, stops
   d) s forwards the request (r, p_c) to the environment

2. a) s receives a response (r, p_s) from the environment
   b) s checks whether the response is admissible and if not, stops
   c) s sends $\{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)\}_{sk_s}$ to c
   d) c checks whether the message is admissible and if not, stops
   e) c forwards the response p_s to the environment

Here, r is the nonce as described in the previous section, which is also used as a handle by the server (see steps 1. d) and 2. a)), t is the value of a local clock of the client, p_c is the payload the client sends, p_s is the payload the server returns, and $\{\cdot\}_{sk_c}$ and $\{\cdot\}_{sk_s}$ stand for signing the message by the client and server, respectively. Repeating the message id of the request allows the client to verify that p_s is indeed a response to the request p_c.

The interesting parts are steps 1. c) and 2. b). We assume that there is a constant $\mathrm{cap}_s > 0$, the so-called capacity of the server, and a constant tol_s^+ that indicates its tolerance with respect to inaccurate clocks. At all times the server keeps a time $t_s^{\min}$ and a finite list L of triples (t, r, c) of pending and handled requests. At the beginning or after a reset, $t^{\min}$ is set to $t_s + \mathrm{tol}_s^+$, where t_s is a timestamp retrieved from the local clock functionality, and L is set to the empty list.

Step 1. c). Upon receiving a message as above, the server s rejects if $t \notin \left[t_s^{\min} + 1, t_s + \mathrm{tol}_s^+\right]$ or if $(t', r, c') \in L$ for some t' and c', and otherwise proceeds

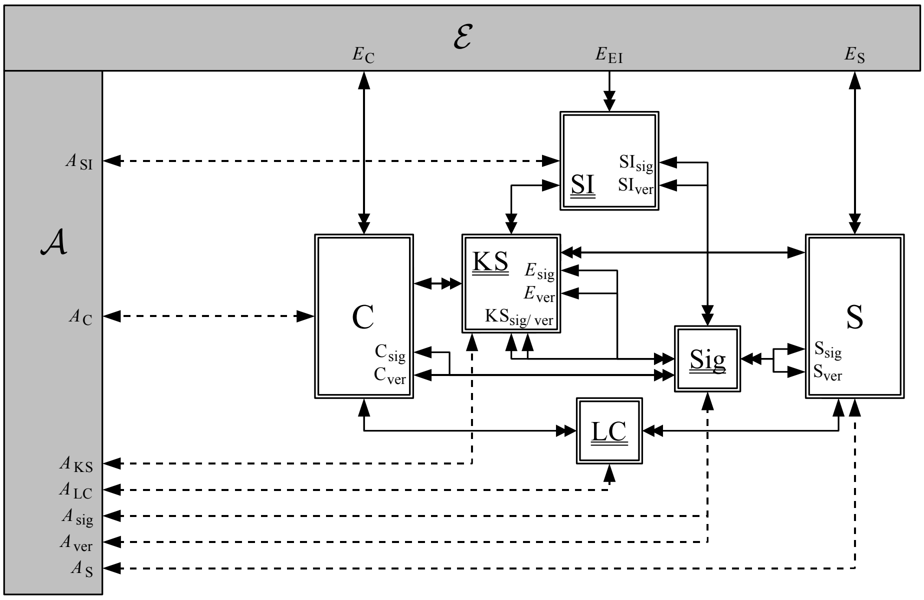

Fig. 4. Overview of 2AMEX-1 protocol implementation

as follows: If L contains less than cap_s elements, it inserts (t, r, c) into L. Otherwise, the server deletes all tuples containing the oldest timestamp from L, until L contains less than cap_s tuples. Then it sets $t_s^{\min}$ to the timestamp contained in the last tuple deleted from L, and finally inserts (t, r, c) into L.

Step 2. b). When asked to send a payload p_s with message handle r, the server rejects if there is no triple $(t, r, c) \in L$ with $c \neq \varepsilon$. If it does not reject, it updates L by overwriting c with ε in the tuple (t, r, c) to ensure that the service cannot respond to the same message twice.

4.2 Implementation in the IITM Model

We will now describe the system of IITM's defined by

$$\mathcal{P}_{2\mathrm{AMEX}-1} = !\mathcal{P}_{\mathrm{C}} \mid !\mathcal{P}_{\mathrm{S}} \mid !\underline{\underline{\mathcal{F}_{\mathrm{Sig}}}} \mid !\underline{\underline{\mathcal{P}_{\mathrm{SI}}}} \mid !\underline{\underline{\mathcal{F}_{\mathrm{KS}}}} \mid !\underline{\underline{\mathcal{F}_{\mathrm{LC}}}} \tag{2}$$

and illustrated in Figure 4, which implements the 2AMEX-1 protocol.

In (2), $\mathcal{P}_{\mathrm{C}}$ is the client-side part of the protocol (defined in Figure 5), $\mathcal{P}_{\mathrm{S}}$ is the server-side part of the protocol (defined in Appendix B.6), $\mathcal{F}_{\mathrm{Sig}}$ is the signature functionality as defined in [KT08b], $\mathcal{P}_{\mathrm{SI}}$ is an interface which allows the adversary to access the signature functionality with few restrictions (defined in Appendix B.7), $\mathcal{F}_{\mathrm{KS}}$ is an ideal functionality of a trusted key store (defined in Appendix B.8), and $\mathcal{F}_{\mathrm{LC}}$ (defined in Appendix B.9) models a local clock which is controlled by the adversary, i. e. not synchronized with the clocks of other parties and not even monotone.

4.3 Signatures and the Public Key Infrastructure

We model the digital signatures that 2AMEX-1 uses by the ideal functionality $\mathcal{F}_{\text{Sig}}$ from [KT08b], which was proved to be securely implementable using any existentially unforgeable signature scheme.

We give the adversary access to the signature scheme and allow him to sign any bit string that does not have the format of a 2AMEX-1 message. This models that our protocol does not have exclusive access to the keys used to sign the messages. For example, the same key can be used to sign a 2AMEX-1 message and parts of the payload contained in that message. This is realized by the signature interface functionality $\mathcal{P}_{\text{SI}}$, which accepts requests from the adversary to (i) sign messages that do not have the format of 2AMEX-1 messages and (ii) verify arbitrary signatures. In $\mathcal{P}_{2\text{AMEX}-1}$, the signature interface functionality is banged in the multi-user multi-session version, effectively meaning that the adversary has access to all keys used in the protocol.

As the signature interface needs resources from the environment to sign messages for the adversary, it has an enriching input tape E_{EI}. Its counterpart in the ideal system is a tape in the enriching input functionality EI.

To coordinate how different IITM's access a single instance of the signature functionality, we define the ideal functionality of a key store, $\mathcal{F}_{\text{KS}}$, which allows clients, servers, and the signature interface functionality to retrieve trusted keys as well as the corruption status of that key. To be able to distribute the public key, $\mathcal{F}_{\text{KS}}$ also initializes the instances of the signature functionality. The particular form of this functionality is due to the fact that we want to use $\mathcal{F}_{\text{Sig}}$ from [KT08b] as is. Nevertheless, one can implement $\mathcal{F}_{\text{KS}}$ using standard techniques for building a public key infrastructure: In an implementation, the key store could be a local subroutine which, (i) locally stores and manages a single public/private key pair, and, (ii) when requested to retrieve the public key of another party, fetches that key from a key server and locally checks its validity by using a trust model, e. g., a pre-defined set of certification authorities.

4.4 Client Implementation

The client protocol $\mathcal{P}_{\text{C}}$ (see Figure 5) is a direct implementation of the ideal functionality $\mathcal{F}_{\text{C}}$ with the following changes:

- The messages are transferred over the network (rather than exchanged directly between client and server). This is modeled by writing the messages on an external network tape.
- To secure the request message, the client signs it using a digital signature obtained from an instance of $\mathcal{F}_{\text{Sig}}$ for this session. The server will be able to obtain the public key from the according key store and verify the signature.
- When receiving a response from the server, the signature of that message is verified by the client in the same way.

```
**Tapes:** C $\twoheadleftarrow\!\!\!\longrightarrow$ $E_{\mathrm{C}}$, C $\dashleftarrow\!\dashrightarrow$ $A_{\mathrm{C}}$, C $\longleftarrow\!\!\!\twoheadrightarrow$ KS, C $\longleftarrow\!\!\!\twoheadrightarrow$ LC, $\mathrm{C}_{\mathrm{sig}}$ $\longleftarrow\!\!\!\twoheadrightarrow$ Sig, $\mathrm{C}_{\mathrm{ver}}$ $\longleftarrow\!\!\!\twoheadrightarrow$ Sig
**Initialization:** $c = s = r = \varepsilon$, $n = 0$, $state = \mathsf{Init}$, $cor = \mathsf{false}$
**Steps:** loop
    *Send a request to the server:*
    if $(c', (\mathsf{Client}, s'), \mathsf{Init})$ received from $E_{\mathrm{C}}$
        If $state \neq \mathsf{Init}$, abort. Let $c = c'$ and $s = s'$.
        Generate an $\eta$-bit nonce $r$ randomly, where $\eta$ is the security parameter.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Nonce}, r)$ to $E_{\mathrm{C}}$.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Request}, p_c, 1^{n'})$ from $E_{\mathrm{C}}$, let $n = n'$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{GetKey})$ to KS.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{PublicKey}, k_c)$ from KS.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{GetTime})$ to LC.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Time}, t)$ from LC.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Corrupted?})$ to KS.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Corrupted}, cor')$ from KS. If $cor'$, abort.
        Let $m_c = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Sign}, m_c)$ on $\mathrm{C}_{\mathrm{sig}}$.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Signature}, \sigma_c)$ on $\mathrm{C}_{\mathrm{sig}}$. Let $state = \mathsf{OK}$.
        Send $(m_c, \sigma_c)$ to $A_{\mathrm{C}}$.
    *Receive and process a response from the server:*
    if $(m_s, \sigma_s)$ received from $A_{\mathrm{C}}$ with $m_s = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)$
        If $state \neq \mathsf{OK}$ or $cor$ or $|p_s| > n$, abort.
        Let $n = n - |p_s|$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{GetKey})$ to KS.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{PublicKey}, k_s)$ from KS.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Init})$ on $\mathrm{C}_{\mathrm{ver}}$.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Init})$ on $\mathrm{C}_{\mathrm{ver}}$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Corrupted?})$ to KS.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Corrupted}, cor')$ from KS. If $cor'$, abort.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Verify}, m_s, \sigma_s, k_s)$ on $\mathrm{C}_{\mathrm{ver}}$.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{Client}, \mathsf{Verified}, b)$ from on $\mathrm{C}_{\mathrm{ver}}$, if $b \neq 1$, stop.
        Let $state = \mathsf{Stopped}$ and send $(c, (\mathsf{Client}, s, r), \mathsf{Response}, p_s)$ to $E_{\mathrm{C}}$.
**Corruption:** $\mathtt{Corr}(cor, \mathsf{true}, state \neq \mathsf{Init}, \varepsilon, A_{\mathrm{C}}, \{E_{\mathrm{C}}\}, E_{\mathrm{C}}, c, (\mathsf{Client}, s, r))$
**CheckAddress:** Check for $c$, $s$, and $r$ as soon as each one has been set.
```

Fig. 5. The client protocol $\mathcal{P}_{\mathrm{C}}$

- The nonces are not generated by a centralized entity, but randomly chosen locally by each client. While this does not guarantee that the numbers are unique, the probability of a collision is negligible if the length of the nonces grows linearly with the security parameter.
- The request message is additionally secured by a timestamp. The client uses the local clock functionality $\mathcal{F}_{\mathrm{LC}}$ to obtain a timestamp.
- Before using a signature functionality to sign or verify a message, the client checks if the signature or the verification functionality is corrupted. If either one is, the client aborts.

4.5 Server Implementation

The implementation $\mathcal{P}_{\mathrm{S}}$ of the server functionality (see Appendix B.6) is more complicated than the client. To be able to counteract replay attacks, one single IITM handles all sessions. That is, for each identity s all communication of that identity in the server role is handled by one single instance of $\mathcal{P}_{\mathrm{S}}$.

Therefore, the server maintains two lists: R stores resources passed by the environment (corresponding to the fact that in the ideal system, each session of the server is started by the environment), while L (corresponding to L described in Section 4.1) is used to store information from request messages received so far by this server. During initialization, i. e., when receiving the first message, the server asks the adversary to provide values for two parameters of the 2AMEX-1 protocol, namely the capacity cap_s and the tolerance tol_s^+.

When receiving a message from the client, the server (i) tries to retrieve the client's key, (ii) obtains the current time from $\mathcal{F}_{\mathrm{LC}}$ (and checks for monotonicity of the clock), (iii) verifies the signature, (iv) checks if a message with the same nonce has already been accepted (i. e. the nonce is in L), (v) checks if the timestamp is in order (i. e. not too old and not too new), and (vi) forwards the message to the environment if everything is in order. If some step fails, the server simply drops the message.

When the environment wants to reply to a message, the server first checks if the nonce is valid (i. e. occurs in L), else it sends an error message to the environment. This is important as the nonce may have been removed from L due to capacity reasons without notification to the environment. Then, the server initializes its instance of the signature scheme for this session, signs the message, and writes it on an external network tape.

Note that during the steps to process a request or a response, the control may be passed to the adversary by some of the ideal functionalities the server uses. Hence, the execution of the steps when processing a request or response may be interrupted by the adversary (e. g., by sending another incoming message to this server). As soon as a message is received that is not related to processing the current message, the processing of the current message is aborted by the server and cannot be resumed later.

5 Results

Our result states that our protocol securely realizes the ideal functionality $\mathcal{F}_{\mathrm{2AM}}$. The formal statement of the theorem is as follows:

Theorem 1.

$$\mathcal{F}_{\mathrm{2AM}} \geq^{\mathrm{BB}} \mathcal{P}_{\mathrm{2AMEX-1}} \geq^{\mathrm{BB}} \mathcal{P}^{\mathrm{JS}}_{\mathrm{2AMEX-1}}$$

where

$$\begin{aligned}
\mathcal{F}_{\mathrm{2AM}} &= !\mathcal{F}_{\mathrm{C}} \mid !\mathcal{F}_{\mathrm{S}} \mid \mathcal{F}_{\mathrm{NG}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{EI}}}}\ , \\
\mathcal{P}_{\mathrm{2AMEX-1}} &= !\mathcal{P}_{\mathrm{C}} \mid !\mathcal{P}_{\mathrm{S}} \mid \underline{\underline{!\mathcal{P}_{\mathrm{SI}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{Sig}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{LC}}}}\ , \\
\mathcal{P}^{\mathrm{JS}}_{\mathrm{2AMEX-1}} &= !\mathcal{P}_{\mathrm{C}} \mid !\mathcal{P}_{\mathrm{S}} \mid \underline{\underline{!\mathcal{P}_{\mathrm{SI}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}}}} \mid !\mathcal{P}^{\mathrm{JS}}_{\mathrm{Sig}} \mid \underline{!\mathcal{F}_{\mathrm{Sig}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{LC}}}}\ .
\end{aligned}$$

Before we give the proof of the theorem, we first explain the involved simulation statements. The first of these inequalities states that the IITM realization of our protocol, when using an ideal signature functionality, realizes the system consisting of the ideal functionalities for $\mathcal{F}_{\mathrm{2AM}}$.

Due to the way in which the ideal signature functionality is used, the realization of the protocol as stated in the first inequality is unrealistic, because for each message sent a new key for the signature scheme is generated. This can be avoided by applying a joint-state theorem [KT08a] allowing different sessions to use the same key. Essentially, a "wrapper" $\mathcal{P}^{\mathrm{JS}}_{\mathrm{Sig}}$ managing different sessions is used to access the signature functionalities. The second inequality in Theorem 1 (which follows directly from [KT08b]) makes use of this wrapper, so that instead of one key per party and per session $(!\underline{\underline{\mathcal{F}_{\mathrm{Sig}}}})$, there is only a single key for each party $(!\underline{\mathcal{F}_{\mathrm{Sig}}})$, as in a realistic public-key infrastructure.

Theorem 1 gives a security treatment of a complex protocol in a simulation-based security setting: Our protocol features a long-lived server role, uses timestamps to prevent replay attacks, and accesses a public-key infrastructure for digital signatures. It is easy to see that long-livedness and timestamps are required to realize our ideal functionality with bounded memory (see [KSW09]). It is interesting to note that while our ideal server functionality is short-lived, a realization necessarily needs to be long-lived; this is a particular property of authenticated message exchange with only two rounds.

We now prove the theorem. A full formal proof would need to establish a bisimulation between the system consisting of the real protocol and that consisting of the ideal protocol and the simulator; the proof below argues why the key points in a correctness proof of the bisimulation can be carried out.

Proof. As mentioned above, it suffices to show the first simulation, as the second one follows directly from [KT08a]. First note that in the ideal functionality $\mathcal{F}_{\mathrm{2AM}}$, we may remove the global nonce generator $\mathcal{F}_{\mathrm{NG}}$ and let each client generate the nonce locally—since the probability of a collision is negligible in the security parameter, the resulting system is computationally indistinguishable from $\mathcal{F}_{\mathrm{2AM}}$. Hence we only need to show that $\mathcal{P}_{\mathrm{2AMEX-1}}$ correctly realizes the thus-modified $\mathcal{F}_{\mathrm{2AM}}$. For the remainder of the proof, when we speak of $\mathcal{F}_{\mathrm{2AM}}$ we mean this modified version.

To prove the theorem, we construct a simulator $\mathcal{S}$ such that the systems $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{F}_{\mathrm{2AM}}$ and $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}_{\mathrm{2AMEX-1}}$ are computationally indistinguishable for every adversary $\mathcal{A}$ and every environment $\mathcal{E}$. The main idea of the simulator (which is presented in Appendix C in detail) is that while interacting with $\mathcal{E}$, $\mathcal{A}$, and all machines that are active in the ideal functionality $\mathcal{F}_{\mathrm{2AM}}$, it simulates every machine that would be present in a run of the system $\mathcal{P}_{\mathrm{2AMEX-1}}$ in such a way that the environment receives the exact same messages on the I/O interface from the machines in $\mathcal{F}_{\mathrm{2AM}}$ as it would receive from the machines in $\mathcal{P}_{\mathrm{2AMEX-1}}$, and analogously presents network traffic to $\mathcal{A}$ that is identical to the traffic a real instance of $\mathcal{P}_{\mathrm{2AMEX-1}}$ would generate on the same inputs.

The key point of the proof is that in our protocol and ideal functionality is that even in the ideal functionality, the adversary may completely control whether a message sent by an instance will reach the environment—hence the simulator essentially consists of book-keeping and allowing the delivery of messages by the ideal functionality as soon as delivery happens in the simulated real functionality.

To show that this simulation indeed works as intended, we argue that for every sequence of messages sent by $\mathcal{A}$ or $\mathcal{E}$, the simulation is correct in the following sense: The state of each simulated machine of the protocol $\mathcal{P}_{\text{2AMEX}-1}$ (i. e., the client machines, server machines, signature functionality, signature interface, and key store) is identical in the simulation and in a hypothetical execution of the real protocol (with the same inputs). We argue separately for each type of machine.

Signature functionality $\mathcal{F}_{\text{Sig}}$. By construction of the simulator, the signature functionality is simulated exactly as it is. It also follows from the discussion below of the (simulated) server and client machines that the simulated signature functionality receives the exact same incoming requests in a real execution of the protocol and in a simulation. Note that resources obtained from the environment for $\mathcal{P}_{\text{SI}}$ are forwarded to the simulated $\mathcal{P}_{\text{SI}}$ directly.

Server protocol machine $\mathcal{P}_{\text{S}}$. By construction, the simulator uses an adaption of the program of the real protocol machine $\mathcal{P}_{\text{S}}$. The negotiation of the initial parameters of the server is directly forwarded to the adversary $\mathcal{A}$, hence the obtained parameters are as in a real execution of the protocol. Note that in a real protocol run, when the server receives a new message while waiting for a reply of the key store functionality or for a signature verification, the waiting is aborted and only the new message is processed—this is mirrored in the simulation by the instruction to cancel currently running jobs for a server when it receives a new message.

By design of the simulation, if a network message is rejected by the server (due to either a false signature, or an outdated timestamp), the state of the server is not changed, and no reply of any kind is sent. Hence in this case the simulated server behaves in the same way as in a real execution of the protocol. In the case that a message is accepted, the list L is maintained as in the real protocol. Instead of notifying the environment about the delivery of the message (as the real protocol implementation would do), the simulator then instructs the (ideal) client to deliver the message to the (ideal) server, which leads to the exact same output to the environment as a delivery to the real server would.

When the environment instructs the (ideal) server to send a reply to a client, then by design of the ideal server functionality, the server asks the adversary whether to proceed. Since $\mathcal{S}$ receives the corresponding query intended for the adversary, it can check whether in the simulated real server, the request of the environment could still be fulfilled (which is the case if and only if a message with the corresponding message id is still present in the list L and has not been marked as answered), and in this case allow the server to proceed.

Note that the simulator simulates the exact same requests made by a server to the signature functionality, hence the simulated functionality receives the exact same messages as it would in a run of the real protocol.

Client protocol machine $\mathcal{P}_{\text{C}}$. This works in much the same way as the server machine: The simulator performs the same verification steps that the real client machine would, and outputs the same data to the environment. Again, the requests for the simulated signature functionality and key store are identical in the simulated and in a real run of the protocol.

Signature interface functionality and key store. As mentioned above, in both real and simulated protocol runs, the signature interface and key store functionalities perform the exact same requests: By construction of the simulator, $\mathcal{A}$ may communicate directly with the simulated machines in the same way as it would in a real protocol run. Since the simulator uses the code of the ideal functionalities, this implies that they are in the same state.

Corruption. By design, a running copy of an ideal client or server functionality is corrupted if and only if a copy of the real server or client would be in a real protocol run. Note that the simulator ensures that as soon as a single copy of a (short-lived) ideal server instance for an identity s is corrupted, then every newly started ideal server instance for this identity is corrupted immediately by the simulator; this mirrors the corresponding behavior in a run of the real protocol, where each identity only a single server machine is running. Hence requests of the form Corrupted? issued by $\mathcal{E}$ get answered positively in the simulated protocol run if and only if the answer would be positive in a real one. Also, the communication with corrupted parties is handled using the same `Corr` macro in the same way in both simulated and real protocol runs, hence the replies of the relevant parties are identical.

6 Discussion

Simulation-based security clearly has the advantage that it leads to an easier statement of security than an individual, trace-based definition, and moreover, allows to treat protocols for very different tasks in a single model. The security properties obtained by such an analysis are quite strong and hold (via composition) in an arbitrary context. The IITM framework (and related frameworks) is designed to support modular protocol analysis.

However, these advantages come with a price when considering a concrete complex protocol. In [KSW09], we presented a customized model (based on the seminal work by Bellare and Rogaway [BR93]) for proving security of 2AMEX-1. A comparison between that work and the current paper gives insights into the advantages and disadvantages of both approaches.

The formulation of both ideal functionalities and concrete implementations for authenticated message exchange in the current paper is rather long and unintuitive (the latter are significantly more complex than their counterparts in [KSW09]). Both feature unnatural communication (bit strings to provide computing resources, status and activation messages exchanged sent to and received from the adversary and the environment), which are necessary due to how resources and activation are handled. Intuitively, one would like the environment to only access the "service" provided by the functionalities, but in the IITM framework, the environment additionally needs to provide resources for the involved parties that allow them to process the input.

Furthermore, the handling of corruption in the IITM framework is more complex and seems less natural than in the Bellare-Rogaway based model. Also, for the analysis of our protocol, the modular approach provided by the IITM framework does not simplify the security analysis, compared to the proof in [KSW09]. Finally, the use of the joint-state theorem to enable realistic treatment of signatures results in a slightly different protocol from the one originally stated in [KSW09] and from a realistic implementation.

It would be very interesting to know whether the IITM framework can be adapted to remove the above-mentioned difficulties.

References

BCJ$^+$06. Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically sound security proofs for basic and public-key Kerberos. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 362–383. Springer, 2006.

BP06. Michael Backes and Birgit Pfitzmann. On the cryptographic key secrecy of the strengthened Yahalom protocol. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, *SEC*, volume 201 of *IFIP*, pages 233–245. Springer, 2006.

BPW04. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A general composition theorem for secure reactive systems. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2004.

BR93. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In D. Stinson, editor, *Advances in Cryptology – Crypto '93, 13th Annual International Cryptology Conference*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 1993.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

CK02. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002.

GMP$^+$08. Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of TLS. In Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai, editors, *ProvSec*, volume 5324 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2008.

KSW09. Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. Computationally secure two-round authenticated message exchange. Cryptology ePrint Archive, Report 2009/262, 2009. `http://eprint.iacr.org/`.

KT08a. Ralf Küsters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. In *CSF*, pages 270–284. IEEE Computer Society, 2008.

KT08b. Ralf Küsters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. Cryptology ePrint Archive, Report 2008/006, 2008. `http://eprint.iacr.org/`.

Küs06. Ralf Küsters. Simulation-based security with inexhaustible interactive Turing machines. In *CSFW*, pages 309–320. IEEE Computer Society, 2006.

LB07. Canyang Kevin Liu and David Booth. Web services description language (WSDL) version 2.0 part 0: Primer. W3C recommendation, W3C, 2007. `http://www.w3.org/TR/wsdl20-primer`.

ML07. Nilo Mitra and Yves Lafon. SOAP version 1.2 part 0: Primer (second edition). Technical report, W3C, 2007. `http://www.w3.org/TR/soap12-part0/`.

MN06. Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 373–392. Springer, 2006.

PW01. Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, pages 184–201, 2001.

Sun98. Sun Microsystems. RPC: Remote procedure call protocol specification version 2. IETF RFC 1057 (Informational), 1998.

Win99. Dave Winer. XML-RPC specification. http://www.xmlrpc.com/spec, 1999.

A Simulation-Based Security

Simulation-based security allows to analyze cryptographic protocols such that properties proven remain true even when the protocol is used as a sub-protocol of a larger system. The main idea is to define a so-called *ideal functionality*, which specifies a cryptographic goal to be realized by a protocol. This ideal functionality also documents the capabilities of an attacker on the protocol. A concrete protocol is “secure” if it *realizes* the ideal functionality such that every attacker on the real protocol can be “simulated” in the ideal setting. We briefly sketch Küsters' model using inexhaustible interactive Turing machines (IITM's). For precise definitions and background on these notions, see [Küs06].

A.1 Inexhaustible Interactive Turing Machines

Cryptographic protocols are modeled as a set of concurrently running machines, called a *system of IITM's* (see below). The machines in the system are activated sequentially, where at each point in time, only a single machine is active, and each machine may be activated repeatedly. A single IITM is a probabilistic Turing machine with an associated polynomial q used to bound its running time and output length. In addition to work tapes, an IITM has named external tapes which may be shared with other machines running concurrently. External read-tapes of machines are partitioned into *consuming* and *enriching* tapes. This distinction serves to allow the maximal running time of the machines to depend on the input on the enriching tapes, and not merely on the security parameter alone as in standard cryptographic models as [BR93]. In order to avoid “exponential blow-up” of lengths of exchanged messages, a *well-formed* system is defined to be one where the sub-graph of machines connected with enriching tapes is acyclic. As proved in [Küs06], a well-formed system can be simulated on a single polynomial-time machine.

External tapes are partitioned into *network tapes* and *I/O-tapes.* The former are used to model communication with subprocesses (here an attacker on the system cannot interfere), the latter model network communication (this is assumed to be controlled by the adversary completely).

An IITM can run in two different modes (determined by the content of the mode tape upon activation): The CheckAddress mode is used to determine whether an incoming message is intended for the current machine. When activated in this mode, the IITM reads an input message from a special input tape and returns accept or reject on a special output tape. In this mode, computation may not be probabilistic, and the number of steps taken must be bounded by $q(n)$, where q is the polynomial associated with the machine, and n is the length of the content of the work tapes, the current input, and the security parameter. This mode is typically used to verify whether an incoming message belongs to the correct session. The Compute mode is then used for the actual computation (which may include replying to the incoming message). The number of steps in this mode must be bounded by $q(n)$, where q and n are as in mode CheckAddress. Additionally, the total output up to a point in the run of the machine, as well as the length of all work tapes must always be bounded by $q(m)$, where m is the sum of the security parameter plus the length of all input received on enriching input tapes in mode Compute in the current run of the system. This implies that when a machine is required to produce "long" output, it previously must be given the corresponding resources via enriching input tapes.

In each activation, a machine produces output on at most one output tape, the machine that has the corresponding tape as an input tape is then activated next. If no output is produced, the *environmental machine* is activated (see below).

A.2 System of IITM's for Cryptographic Protocols

A *system of IITM's* is an expression of the form

$$\mathcal{P} = M_1 \parallel \ldots \parallel M_k \parallel !M'_1 \parallel \ldots \parallel !M'_k \ , \tag{3}$$

where the M_i and M'_i are IITM's. The machines $M'_1, \ldots, M'_k$ are said to appear in the *scope of a bang*: The bang operator "!" provides an "infinite supply" of machines (running the code of) M'_i. In a run of a system, this is handled as follows: When a machine M sends a message (via a shared tape) to a machine M' of which a copy is already running, but this copy rejects the message in its CheckAddress mode and M' appears in the scope of a bang, then a new instance of M' is started, which then may accept the message in CheckAddress mode. If it does, it remains active and processes the incoming message. Otherwise it is deactivated again. This allows to start an unbounded number of sessions of a protocol.

An *external* tape of a system $\mathcal{P}$ is a tape which is a network- or I/O-tape of one of its machines for which there is no corresponding output or input tape in the system itself. These tapes allow external machines to communicate with $\mathcal{P}$, and thus enable $\mathcal{P}$ to provide a functionality to "outside" machines. This mechanism allows to naturally compose systems of IITM's in a way allowing interaction: For two systems $\mathcal{P}_1$ and $\mathcal{P}_2$, $\mathcal{P}_1 \mid \mathcal{P}_2$ denotes the system containing all machines of $\mathcal{P}_1$ and $\mathcal{P}_2$, where internal tapes of the systems are consistently

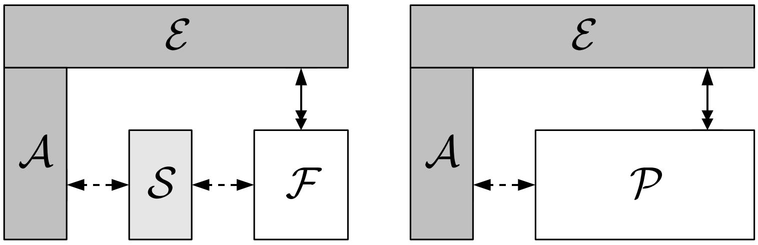

Fig. 6. An abstract view of the two systems of IITM's

renamed (the systems only influence each other via their communication on their external tapes).

To define security notions for cryptographic protocols, the composition of a given system with an *environment* and an *adversary* are studied. An *adversary* for $\mathcal{P}$ is a system $\mathcal{A}$ such that the set of external I/O-tapes of $\mathcal{P}$ and $\mathcal{A}$ are disjoint, and for every external output network tape of $\mathcal{P}$, there is an external network input tape of $\mathcal{A}$, and vice versa. This means that an adversary for $\mathcal{P}$ is syntactically suited to connect to all external "network ports" of $\mathcal{P}$. Typically, all incoming external tapes of an adversary are defined to be enriching. An *environmental* system for $\mathcal{P}$ similarly connects to the I/O-tapes, and its set of external network tapes is disjoint with that of $\mathcal{P}$. When $\mathcal{P}$ and $\mathcal{F}$are systems (the real and the ideal system), then an adversarially connectable system $\mathcal{S}$ is a *simulator for $\mathcal{F}$and $\mathcal{P}$*, if $\mathcal{S} \mid \mathcal{F}$ has the exact same set of external tapes (with matching type and direction) as $\mathcal{P}$ (note that an output (input) tape in $\mathcal{S} \mid \mathcal{F}$ is only external when there is no input (output) tape with the same name in $\mathcal{S}$ *or* in $\mathcal{F}$. Hence a simulator only connects to the network tapes of $\mathcal{F}$, and syntactically, $\mathcal{S} \mid \mathcal{F}$ and $\mathcal{P}$ "look the same". In particular, a system $\mathcal{E}$ is a suitable environment for $\mathcal{P}$ if and only if it is one for $\mathcal{S} \mid \mathcal{F}$.

A system may have a special external output tape named decision. When a machine writes output to this tape (the output must be either 0 or 1), the run of the system stops immediately. Two systems are *equivalent*, if the probability that for the same input, a different value is written on the decision tape, is negligible (in the security parameter). This means that for an outside observer that may only interact with the systems using their I/O-interface, the systems behave identically with overwhelming probability.

We now define the central security notion that we study, also see Figure 6—in the following, $\mathcal{F}$ is supposed to be an "ideal" system (also called *ideal functionality*, and $\mathcal{P}$ a concrete system that attempts to "realize" the ideal functionality. $\mathcal{P}$ and $\mathcal{F}$ are I/O-compatible if they have disjoint sets of external network tapes, the same set of external I/O-tapes, and each external I/O-tape has the same direction in both.

Definition 2. *Let $\mathcal{P}$ and $\mathcal{F}$ be I/O-compatible systems. Then $\mathcal{P} \leq^{\mathrm{BB}} \mathcal{F}$.if there is a simulator $\mathcal{S}$ for $\mathcal{P}$ and $\mathcal{F}$ such that for all adversaries $\mathcal{A}$ and environments $\mathcal{E}$ for $\mathcal{P}$ or $\mathcal{S} \mid \mathcal{F}$, the systems $\mathcal{E} \mid \mathcal{P}$ and $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ are equivalent.*

Here "equivalent" means that with overwhelming probability, the same input leads to the same output. This models the intuition expressed above: The simulator $\mathcal{S}$ essentially makes the system $\mathcal{F}$behave exactly as $\mathcal{P}$ (without the simulator). Hence any attack that can be mounted on the real protocol system $\mathcal{P}$ is also successful against the ideal functionality $\mathcal{F}$.

A.3 Session Versions of IITM's

The IITM model offers a simple mechanism for specifying multi-session variants of a protocol: For an IITM M, the machine $\underline{M}$ simulates M, and expects that all incoming messages are prefixed with a session-id. This session-id is then removed from the string actually handed to the simulated M, and is added as a prefix to every message written by the simulated M on an output tape. Hence a system of the form $!\underline{M}$ has an unlimited supply of machines executing the code of M, each using an independent session. Multi-party, multi-session variants of a protocol, are then obtained by using $\underline{\underline{M}}$: These machines handle prefixes containing a party- and a session-id.

B Functionalities and Protocols

B.1 Notation

When defining an IITM M, we describe it in the following way:

First, we define the tapes of M. We denote by $A \leftrightarrow B$ a tape or a pair of tapes in the following way:

- the label on the left-hand side (e. g., A) is the name of the tape on M's side of the tape, whereas the label on the right-hand side (e. g., B) is the name of the tape on the machine that M is connected to,
- a single output tape is denoted by $\longrightarrow$, a single input tape is denoted by $\longleftarrow$, and a pair of input and output tapes is denoted by $\longleftrightarrow$,
- a consuming tape is denoted by $\longrightarrow$, an enriching tape by $\twoheadrightarrow$,
- an I/O tape is denoted by $\longrightarrow$, a network tape by $\dashrightarrow$.

Next, we may define an initialization routine which is executed when the IITM is activated for the first time.

In the main part, we describe a couple of steps: We assume that the machine matches each incoming message against a couple of patterns, executing the first step that has a matching pattern, and discarding the message if no step matches. During the execution of a single step, if the machine waits for a specific message, it will ignore all other incoming messages, even if they would match any of the patterns of the steps.

For the simulator we also define functions or subroutines.

In some functionalities we then include the parameterized corruption macro, see below. This adds a couple of steps which take precedence over the steps we defined above.

Last, for most functionalities we describe the IITM's operation in CheckAddress mode, where the default mode is to accept all incoming messages.

B.2 Message Format

Due to the addressing mechanism used in the IITM model, a message that is being sent to a banged IITM has to contain information allowing all currently running instances to decide which of them is supposed to accept that message.

Thus, our messages have the format $(pid, sid, ...)$ where pid is a party id and sid is a session id. The party id is used to identify a client or a server, usually the sender of the message or, in case the message comes from the environment or the adversary, the recipient of this message. The session id usually consists of three parts: (i) a constant, either Client or Server, to distinguish the role of the party, (ii) the identity of the communication partner, and (iii) the nonce used in this session.

Note that in our case it is not possible to use the identifier version as defined in [KT08a] because of two reasons: Firstly, when initializing a new instance, at least the nonce is not yet known by the initializing party (i. e., the environment). Secondly, the parties have to communicate with different pids and sids than their own, i. e., while communicating with server s and using nonce r, a client c has to access both the key with pid c and sid (Client, s, r) for signing as well as the key with pid s and sid (Server, c, r) for verifying.

B.3 Corruption

Both in the ideal functionality $\mathcal{F}_{\mathrm{2AM}}$ and in the implementation $\mathcal{P}_{\mathrm{2AMEX-1}}$ we model corruption by using the corruption macro from [KT08a] in a slightly modified variant, in which we add parameters for an addressing mechanism. The modified macro is defined in Appendix B.10.

Using the corruption macro we allow the adversary to corrupt our clients and servers, while the environment can check the corruption status of each instance and provide resources for corrupted machines. Once corrupted, clients and servers abort their normal execution and only forward messages from and to the adversary as defined in the macro.

While the adversary can corrupt single client instances, the situation on the server side is different: If the adversary sends a corruption request to one instance of $\mathcal{F}_{\mathrm{S}}$ running under identity s, this instance will accept all messages which are directed to any instance running under identity s. This reflects that in the implementation $\mathcal{P}_{\mathrm{2AMEX-1}}$ only one (long-lived) instance of $\mathcal{P}_{\mathrm{S}}$ is running per identity.

Note that the signature and verification functionality $\mathcal{F}_{\mathrm{Sig}}$ used in $\mathcal{P}_{\mathrm{2AMEX-1}}$ also allows corruption. But if the adversary would corrupt, e. g., a verification instance, it would have no advantage against our protocol as long as it does not also corrupt the server or client using that particular instance of the verifier. In addition, in $\mathcal{P}_{2\mathrm{AMEX}-1}$ the environment would have to pass resources to that verification instance, while in $\mathcal{F}_{2\mathrm{AM}}$ no signature scheme is available to receive the resources—but adding a mechanism to $\mathcal{F}_{2\mathrm{AM}}$ which receives the resources and passes them on to the simulator would result in a rather unnatural ideal functionality.

Therefore, even though we technically allow the adversary to corrupt instances of the signature scheme (or its verifiers) in $\mathcal{P}_{2\mathrm{AMEX}-1}$, we make it rather useless: Before $\mathcal{P}_\mathrm{C}$ and $\mathcal{P}_\mathrm{S}$ use any signature or verification functionality, they check the functionalities' corruption status and abort if it is corrupted. Note that the adversary may still get complete control over the input and output of a client or server by simply corrupting that client or server instance.

B.4 The Nonce Generator Functionality $\mathcal{F}_{\mathbf{NG}}$

```
Tapes: $\mathrm{NG} \twoheadleftarrow\!\!\!\longrightarrow \mathrm{C}$
Initialization: $L = [\,]$
Steps: loop
    Generate a fresh nonce:
    if $(\mathit{pid}, \mathit{sid}, \mathsf{GetNonce}) = m$ received from C
        Generate an $\eta$-bit nonce $r$ randomly with $r \notin L$, as long as $|L| \leq 2^\eta$,
          where $\eta$ is the security parameter.
        Insert $r$ in $L$.
        Send $(\mathit{pid}, \mathit{sid}, \mathsf{Nonce}, r)$ to C.
```

B.5 The Enriching Input Functionality $\mathcal{F}_{\mathbf{EI}}$

```
Tapes: $\mathrm{EI} \twoheadleftarrow E_\mathrm{EI}$, $\mathrm{EI} \dashleftarrow\!\dashrightarrow \hat{A}_\mathrm{EI}$
Steps: loop
    Forward resources:
    if $(\mathsf{Resources}, 1^n, b)$ received from $E_\mathrm{EI}$
        Send $(\mathsf{Resources}, b, n)$ to $\hat{A}_\mathrm{EI}$.
```

B.6 The Server Protocol $\mathcal{P}_{\mathbf{S}}$

```
Tapes: $\mathrm{S} \twoheadleftarrow\!\!\!\longrightarrow E_\mathrm{S}$, $\mathrm{S} \dashleftarrow\!\dashrightarrow A_\mathrm{S}$, $\mathrm{S} \longleftarrow\!\!\!\twoheadrightarrow \mathrm{KS}$, $\mathrm{S} \longleftarrow\!\!\!\twoheadrightarrow \mathrm{LC}$, $\mathrm{S_{sig}} \longleftarrow\!\!\!\twoheadrightarrow \mathrm{Sig}$, $\mathrm{S_{ver}} \longleftarrow\!\!\!\twoheadrightarrow \mathrm{Sig}$
Initialization: $s = \mathrm{cap}_s = \mathrm{tol}_s^+ = m_c = \sigma_c = k_c = \varepsilon$, $R = L = [\,]$, $t_s = t^\mathrm{min} = 0$, $\mathit{state} = \mathsf{Init}$,
    $\mathit{cor} = \mathsf{false}$
Steps: loop
    Initialize a new buffer:
    if $(s', (\mathsf{Server}), \mathsf{Init}, 1^n)$ received from $E_\mathrm{S}$
        If $\mathit{state} = \mathsf{Init}$,
          Send $(s', (\mathsf{Server}), \mathsf{GetParameters})$ to $A_\mathrm{S}$.
          Recv $(s', (\mathsf{Server}), \mathsf{Parameters}, \mathrm{cap}, \mathrm{tol}^+)$ from $A_\mathrm{S}$.
          Let $s = s'$. If $\mathrm{cap} \leq 0$ or $\mathrm{tol}^+ \leq 0$, abort.
          Send $(s, (\mathsf{Server}, c, r), \mathsf{GetTime})$ to LC.
          Recv $(s, (\mathsf{Server}, c, r), \mathsf{Time}, t)$ from LC.
          Let $\mathit{state} = \mathsf{OK}$, $\mathrm{cap}_s = \mathrm{cap}$, $\mathrm{tol}_s^+ = \mathrm{tol}^+$, $t_s = t$, $t^\mathrm{min} = t_s + \mathrm{tol}_s^+$.
        Append $n$ to $R$.
    Receive and process a request: Request the client's key:
    if $(m, \sigma)$ received from $A_\mathrm{S}$ with $m = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)$
        If $\mathit{state} = \mathsf{Init}$ or $R$ is empty or $\mathit{cor}$, abort.
        Let $n$ be the first item of $R$. If $|p_c| > n$, abort. Remove $n$ from $R$.
        Let $\mathit{state} = \mathsf{WaitingForKey}_c$, $m_c = m$, and $\sigma_c = \sigma$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{GetKey})$ to KS.
    Receive and process a request: Receive the key, request time:
    if $(c, (\mathsf{Client}, s, r), \mathsf{PublicKey}, k)$ received from KS
        If $state \neq \mathsf{WaitingForKey}_c$ or $cor$, abort. Let $state = \mathsf{WaitingForTime}$ and $k_c = k$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{GetTime})$ to LC.
    Receive and process a request: Receive time, initialize the verifier:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Time}, t)$ received from LC
        If $state \neq \mathsf{WaitingForTime}$ or $cor$, abort.
        If $t \geq t_s$, let $t_s = t$. Let $state = \mathsf{WaitingForVerifier}$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Init})$ on $\mathrm{S}_{\mathrm{ver}}$.
    Receive and process a request: Execute 2AMEX-1 protocol steps, relay request:
    if $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Init})$ received on $\mathrm{S}_{\mathrm{ver}}$
        If $state \neq \mathsf{WaitingForVerifier}$ or $cor$, abort. Let $state = \mathsf{OK}$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Corrupted?})$ to KS.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Corrupted}, cor')$ from KS. If $cor'$, abort.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Verify}, m_c, \sigma_c, k_c)$ on $\mathrm{S}_{\mathrm{ver}}$.
        Recv $(c, (\mathsf{Client}, s, r), \mathsf{Server}, \mathsf{Verified}, b)$ on $\mathrm{S}_{\mathrm{ver}}$.
        If $b \neq 1$, $t \leq t^{\min}$ or $t > t_s + \mathrm{tol}_s^{+}$, or $(t', r, c') \in L$ for some $t', c'$, abort.
        While $|L| \geq \mathrm{cap}_s$:
            Let $t^{\min} = \min\{t' \mid (t', r', c') \in L\}$ and $L = \{(t', r', c') \in L \mid t' > t^{\min}\}$.
        Insert $(t, r, c)$ into $L$ and send $(s, (\mathsf{Server}, c, r), \mathsf{Request}, p_c)$ to $E_{\mathrm{S}}$.
    Receive and process a response: Receive response payload, request key:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p_s)$ received from $E_{\mathrm{S}}$
        If $state = \mathsf{Init}$ or $cor$, abort.
        If $(t', r, c) \notin L$ for any $t'$:
            Let $state = \mathsf{OK}$, send $(s, (\mathsf{Server}, c, r), \mathsf{Response}, \mathsf{Error})$ to $E_{\mathrm{S}}$, and abort.
        Let $state = \mathsf{WaitingForKey}_s$ and send $(s, (\mathsf{Server}, c, r), \mathsf{GetKey})$ to KS.
    Receive and process a response: Construct, sign, and send response message:
    if $(s, (\mathsf{Server}, c, r), \mathsf{PublicKey}, k)$ received from KS
        If $state \neq \mathsf{WaitingForKey}_s$ or $cor$, abort. Let $state = \mathsf{OK}$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Corrupted?})$ to KS.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{Corrupted}, cor)$ from KS. If $cor'$, abort.
        Let $m_s = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)$.
        Send $(s, (\mathsf{Server}, c, r), \mathsf{Sign}, m_s)$ on $\mathrm{S}_{\mathrm{sig}}$.
        Recv $(s, (\mathsf{Server}, c, r), \mathsf{Signature}, \sigma_s)$ on $\mathrm{S}_{\mathrm{sig}}$.
        Update $(t, r, c)$ to $(t, r, *)$ in $L$ and send $(m_s, \sigma_s)$ to $A_{\mathrm{S}}$.
    Reset the server:
    if $(s, \mathsf{Reset})$ received from $A_{\mathrm{S}}$
        If $state = \mathsf{Init}$ or $cor$, abort.
        Send $(s, \mathsf{Server}, \mathsf{GetTime})$ to LC.
        Recv $(s, \mathsf{Server}, \mathsf{Time}, t)$ from LC.
        If $t \geq t_s$, let $t_s = t$.
        Let $t^{\min} = t_s + \mathrm{tol}_s^{+}$, $R = L = [\,]$, and $state = \mathsf{OK}$.
```

Corruption: $\mathtt{Corr}(cor, \mathsf{true}, state \neq \mathsf{Init}, \varepsilon, A_{\mathrm{S}}, \{E_{\mathrm{S}}\}, E_{\mathrm{S}}, s)$

CheckAddress: Check for s as soon as it has been set.

B.7 The Signature Interface Protocol $\mathcal{P}_{\mathbf{SI}}$

Tapes: SI $\twoheadleftarrow E_{\mathrm{EI}}$, SI $\dashleftarrow\!\dashrightarrow A_{\mathrm{SI}}$, SI $\longleftrightarrow\!\!\!\!\twoheadrightarrow$ KS, $\mathrm{SI}_{\mathrm{sig}} \longleftrightarrow\!\!\!\!\twoheadrightarrow$ Sig, $\mathrm{SI}_{\mathrm{ver}} \longleftrightarrow\!\!\!\!\twoheadrightarrow$ Sig

Initialization: $state = \mathsf{Init}$, $res = 0$, $k = \varepsilon$

Steps: `loop`

```
    Get resources from the environment to sign messages:
    if $(\mathsf{Resources}, 1^n)$ received from $E_{\mathrm{EI}}$
        Let $res = res + n$,
        If $state = \mathsf{Init}_0$, let $state = \mathsf{Init}_1$.
    Initialization—initialize the key and the verification functionality:
    if $(\mathsf{Init})$ received from $A_{\mathrm{SI}}$
        If $state \neq \mathsf{Init}_1$, abort.
        Send $(\mathrm{SI}, \mathsf{GetKey})$ to KS.
        Receive $(\mathrm{SI}, \mathsf{PublicKey}, k')$ from KS.
        Let $k = k'$.
        Send $(\mathrm{SI}, \mathsf{Init})$ on $\mathrm{SI}_{\mathrm{ver}}$.
        Receive $(\mathrm{SI}, \mathsf{Init})$ on $\mathrm{SI}_{\mathrm{ver}}$.
        Let $state = \mathsf{OK}$.
        Send $(\mathsf{PublicKey}, k)$ to $A_{\mathrm{SI}}$.
    Sign a message:
    if $(\mathsf{Sign}, m)$ received from $A_{\mathrm{SI}}$
        If $\mathit{state} \neq \mathsf{OK}$, abort.
        If $m \in X$, abort.
        If $|m| > \mathit{res}$, abort.
        Let $\mathit{res} = \mathit{res} - |m|$.
        Send $(\mathsf{Sign}, m)$ on $\mathrm{SI}_{\mathrm{sig}}$.
        Receive $(\mathsf{Signature}, \sigma)$ on $\mathrm{SI}_{\mathrm{sig}}$.
        Send $(\mathsf{Signature}, \sigma)$ to $A_{\mathrm{SI}}$.
    Verify a message:
    if $(\mathsf{Verify}, m, \sigma)$ received from $A_{\mathrm{SI}}$
        If $\mathit{state} \neq \mathsf{OK}$, abort.
        If $|m| > \mathit{res}$, abort.
        Let $\mathit{res} = \mathit{res} - |m|$.
        Send $(\mathrm{SI}, \mathsf{Verify}, m, \sigma, k)$ on $\mathrm{SI}_{\mathrm{ver}}$.
        Receive $(\mathrm{SI}, \mathsf{Verified}, b)$ on $\mathrm{SI}_{\mathrm{ver}}$.
        Send $(\mathsf{Verified}, b)$ to $A_{\mathrm{SI}}$.
```

B.8 The Key Store Functionality $\mathcal{F}_{\mathbf{KS}}$

Tapes: KS $\twoheadleftarrow\!\!\longrightarrow$ SI, KS $\twoheadleftarrow\!\!\longrightarrow$ C, KS $\twoheadleftarrow\!\!\longrightarrow$ S, KS $\dashleftarrow\!\dashrightarrow$ A_{KS}, $\mathrm{KS}_{\mathrm{sig}}$ $\longleftarrow\!\!\twoheadrightarrow$ Sig, $\mathrm{KS}_{\mathrm{ver}}$ $\longleftarrow\!\!\twoheadrightarrow$ Sig, E_{sig} $\longleftarrow\!\!\twoheadrightarrow$ Sig, E_{ver} $\longleftarrow\!\!\twoheadrightarrow$ Sig
Initialization: $k = \varepsilon$, $L_{\mathrm{ToDo}} = [\,]$
Steps: `loop`

```
    Request to get the key:
    if $(\mathsf{GetKey})$ received from $T \in \{\mathrm{C}, \mathrm{S}, \mathrm{SI}\}$
        Insert $T$ into $L_{\mathrm{ToDo}}$.
        Send $(\mathsf{GetKey}, T)$ to $A_{\mathrm{KS}}$.
    Execute request to get the key:
    if $(\mathsf{GetKey}, T)$ received from $A_{\mathrm{KS}}$
        If $T \notin L_{\mathrm{ToDo}}$, abort.
        If $k = *$, send $(\mathsf{Init})$ on $\mathrm{KS}_{\mathrm{sig}}$ and stop.
        Delete $T$ from $L_{\mathrm{ToDo}}$.
        Send $(\mathsf{PublicKey}, k)$ to $T$.
    Store a generated key and notify the adversary:
    if $(\mathsf{PublicKey}, k')$ received on $\mathrm{KS}_{\mathrm{sig}}$
        Let $k = k'$.
        Send $(\mathsf{PublicKey}, k)$ to $A_{\mathrm{KS}}$.
    Is the signature functionality corrupted?
    if $(\mathsf{Corrupted?})$ received from $T \in \{\mathrm{C}, \mathrm{S}, \mathrm{SI}\}$
        Send $(\mathsf{Corrupted?})$ on $E_{\mathrm{sig}}$.
        Receive $(x)$ on $E_{\mathrm{sig}}$.
        Send $(\mathsf{Corrupted}, x)$ to $T$.
    Is the verification functionality corrupted?
    if $(\mathit{id}, \mathsf{Corrupted?})$ received from $T \in \{\mathrm{C}, \mathrm{S}, \mathrm{SI}\}$
        Send $(\mathit{id}, \mathsf{Corrupted?})$ on $E_{\mathrm{ver}}$.
        Receive $(\mathit{id}, x)$ on $E_{\mathrm{ver}}$.
        Send $(\mathit{id}, \mathsf{Corrupted}, x)$ to $T$.
```

B.9 The Local Clock Functionality $\mathcal{F}_{\mathbf{LC}}$

Tapes: LC $\twoheadleftarrow\!\!\longrightarrow$ C, LC $\twoheadleftarrow\!\!\longrightarrow$ S, LC $\dashleftarrow\!\dashrightarrow$ A_{LC}
Steps: `loop`

```
    Forward resources:
    if $(\mathsf{GetTime})$ received from $T \in \{\mathrm{C}, \mathrm{S}\}$
        Send $(\mathsf{GetTime})$ to $A_{\mathrm{LC}}$.
        Recv $(\mathsf{Time}, t)$ from $A_{\mathrm{LC}}$.
        Send $(\mathsf{Time}, t)$ to $T$.
```

B.10 The Modified Corruption Macro Corr

The following corruption macro is a modified version of the one defined in [KT08a]; we added a simple addressing mechanism.

```
Macro Corr($corrupted \in \{\mathsf{true}, \mathsf{false}\}$, $corruptible \in \{\mathsf{true}, \mathsf{false}\}$, $initialized \in \{\mathsf{true}, \mathsf{false}\}$,
    $corrMsg$, $T_{\mathrm{adv}}$, $\mathcal{T}_{\mathrm{user}}$, $T_{\mathrm{env}}$, $id_1, \ldots, id_n$)
Initialization: $res = 0$
Steps: loop
    Corruption Request:
    if $(id_1, ..., id_n, \mathsf{Corrupted?})$ received from $T_{\mathrm{env}}$
        If intialized, send $(corrupted)$ to $T_{\mathrm{env}}$.
    Corruption:
    if $(id_1, ..., id_n, \mathsf{Corrupt})$ received from $T_{\mathrm{adv}}$
        If corruptible, initalized, and not corrupted:
            Let $corrupted = \mathsf{true}$.
            Send $(id_1, ..., id_n, \mathsf{Corrupted}, corrMsg)$ to $T_{\mathrm{adv}}$.
    Forward to A (this rule takes precedence over all other rules):
    if $(id_1, ..., id_n, ...) = m$ received from $T \in \mathcal{T}_{\mathrm{user}}$ and corrupted
        Let $res = 0$ and send $(id_1, ..., id_n, \mathsf{Recv}, m, T)$ to $T_{\mathrm{adv}}$.
    Forward to user:
    if $(id_1, ..., id_n, \mathsf{Send}, m, T)$ received from $T_{\mathrm{adv}}$, $T \in \mathcal{T}_{\mathrm{user}}$, corrupted, $0 < |m| \leq res$, and
      $m = (id_1, ..., id_n, ...)$
        Send $m$ to $T$.
    Ressources:
    if $(id_1, ..., id_n, \mathsf{Resources}, r)$ received from $T_{\mathrm{env}}$ and corrupted
        Let $res = |r|$ and send $(id_1, ..., id_n, \mathsf{Resources}, r)$ to $T_{\mathrm{adv}}$.
CheckAddress: Check for $id_1, ..., id_n$.
```

C Simulator

```
Tapes: SI $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{SI}}$, C $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{C}}$, KS $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{KS}}$, LC $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{LC}}$, sig $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{sig}}$, ver $\twoheadleftarrow\dashrightarrow$
    $A_{\mathrm{ver}}$, S $\twoheadleftarrow\dashrightarrow$ $A_{\mathrm{S}}$, $\hat{A}_{\mathrm{EI}}$ $\leftarrow\dashrightarrow$ EI, $\hat{A}_{\mathrm{C}}$ $\leftarrow\dashrightarrow$ C, $\hat{A}_{\mathrm{S}}$ $\leftarrow\dashrightarrow$ S
Initialization: $c = s = r = \varepsilon$, $n = 0$, $state = \mathsf{Init}$, $cor = \mathsf{false}$
Steps: loop
    Initialization of the server:
    if $(s, (\mathsf{Server}), \mathsf{Init}, n)$ received from S
        If $state[s] \neq \mathsf{Init}$,
            Run processServerInit($s, n$) concurrently.
    Receive a request from the client:
    if $(c, (\mathsf{Client}, s, r), \mathsf{Request}, p_c, n)$ received from C
        Run processClientSend($c, s, r, p_c, n$) concurrently.
    Deliver a request to the server:
    if $(m_c, \sigma_c)$ received from $A_{\mathrm{S}}$ with $m_c = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t_c, \mathsf{Body}\colon p_c)$
        Cancel any concurrent runs of processServerReceive or processServerSend with server
          identity $s$.
        Run processServerReceive($m_c, \sigma_c$) concurrently.
    Receive response from the server:
    if $(s, (\mathsf{Server}, c, r), \mathsf{Response}, p_s)$ received from S
        Cancel any concurrent runs of processServerReceive or processServerSend with server
          identity $s$.
        Run processServerSend($s, c, r, p_s$) concurrently.
    Deliver a response to the client:
    if $(m_s, \sigma_s)$ received from $A_{\mathrm{C}}$ with $m_s = (\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)$
        Run processClientReceive($m_s, \sigma_s$) concurrently.
    Reset the server:
    if $(s, \mathsf{Reset})$ received from $A_{\mathrm{S}}$
        Cancel any concurrent runs of processServerReceive or processServerSend with server
          identity $s$.
        Run processServerReset($s$) concurrently.
    Corruption Request:
    if $(id_1, ..., id_n, \mathsf{Corrupt})$ received from $A_{\mathrm{C}}$, $A_{\mathrm{S}}$, $A_{\mathrm{sig}}$, or $A_{\mathrm{ver}}$
        processCorruptionRequest($id_1, ..., id_n, T$).
```

```
Corrupted forward to the adversary:
if $(id_1, ..., id_n, \mathsf{Recv}, m, T)$ received from C or S
        Send $(id_1, ..., id_n, \mathsf{Recv}, m, T)$ to $A_{\mathrm{C}}$ or $A_{\mathrm{S}}$.
Corrupted forward to the user:
if $(id_1, ..., id_n, \mathsf{Send}, m, T)$ received from $A_{\mathrm{C}}$ or $A_{\mathrm{S}}$
        Send $(id_1, ..., id_n, \mathsf{Send}, m, T)$ to C or S.
Ressources for Signing:
if $(pid, sid, \mathsf{Resources}, 1^n)$ received from EI
        Send $(pid, sid, \mathsf{Resources}, 1^n)$ to SI.
In addition, simulate $!\underline{\underline{\underline{\mathcal{F}_{\mathrm{Sig}}}}} \mid !\underline{\underline{\underline{\mathcal{P}_{\mathrm{SI}}}}} \mid !\underline{\underline{\underline{\mathcal{F}_{\mathrm{KS}}}}} \mid !\underline{\underline{\underline{\mathcal{F}_{\mathrm{LC}}}}}$ and answer internal requests as well as request
      from the adversary to these machines.
```

Functions:

```
Initialization of the server:
processServerInit($s, n$)
        If $state[s] = \varepsilon$,
           Send $(s, (\mathsf{Server}), \mathsf{GetParameters})$ to $A_{\mathrm{S}}$.
           Recv $(s, (\mathsf{Server}), \mathsf{Parameters}, \mathrm{cap}, \mathrm{tol}^+)$ from $A_{\mathrm{S}}$.
           If $\mathrm{cap} \leq 0$ or $\mathrm{tol}^+ \leq 0$, abort.
           Let $t = \mathtt{getTime}(s, (\mathsf{Server}, c, r))$.
           Let $state[s] = \mathsf{OK}$, $\mathrm{cap}[s] = \mathrm{cap}$, and $\mathrm{tol}^+[s] = \mathrm{tol}^+$.
           Let $t[s] = t$, $t^{\min}[s] = t[s] + \mathrm{tol}^+[s]$, and $R[s] = L[s] = [\,]$.
           Let $state[s] = \mathsf{Init}$.
        Append $n$ to $R[s]$.
        If $cor[\mathsf{Server}, s]$,
           corruptServer($s$)
           Let $state[s] = \mathsf{OK}$.
        Send $(s, (\mathsf{Server}), \mathsf{Init}, \mathsf{OK})$ to S.
Receive a request from the client:
processClientSend($c, s, r, p_c, n$)
        Let $state[c, s, r] = \mathsf{OK}$ and $n[c, s, r] = n$.
        Let $k = \mathtt{getKey}(c, (\mathsf{Client}, s, r))$.
        Let $t = \mathtt{getTime}(c, (\mathsf{Client}, s, r))$.
        Let $m_c = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)$.
        Let $\sigma_c = \mathtt{sign}(c, (\mathsf{Client}, s, r), m_c)$.
        Send $(m_c, \sigma_c)$ to $A_{\mathrm{C}}$.
Deliver a request to the server:
processServerReceive($m_c, \sigma_c$)
        Decode $m_c$ into $(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t_c, \mathsf{Body}\colon p_c)$.
        If $state[s] \neq \mathsf{OK}$, $cor[\mathsf{Server}, s]$, or $R[s]$ is empty, abort.
        Let $n$ be the first item of $R[s]$. If $|p_c| > n$, abort. Remove $n$ from $R[s]$.
        Let $k = \mathtt{getKey}(c, (\mathsf{Client}, s, r))$.
        Let $t' = \mathtt{getTime}(s, (\mathsf{Server}, c, r))$.
        If $t' \geq t[s]$, let $t[s] = t'$.
        Let $b = \mathtt{verify}(c, (\mathsf{Client}, s, r), \mathsf{Server}, m_c, \sigma_c, k)$.
        If $b \neq 1$, $t_c \leq t^{\min}[s]$ or $t_c > t[s] + \mathrm{tol}^+_s$, or $(t', r, c') \in L[s]$ for some $t', c'$, abort.
        While $|L[s]| \geq \mathrm{cap}_s$:
           Let $t^{\min}[s] = \min\{t' \mid (t', r', c') \in L[s]\}$.
           Let $L[s] = \{(t', r', c') \in L[s] \mid t' > t^{\min}[s]\}$.
        Insert $(t, r, c)$ into $L[s]$.
        Send $(c, (\mathsf{Client}, s, r), \mathsf{Request}, \mathsf{Send})$ to C.
Receive response from the server:
processServerSend($s, c, r, p_s$)
        If $state[s] \neq \mathsf{OK}$, abort.
        If $(t', r, c) \notin L[s]$ for any $t'$,
           Send $(s, (\mathsf{Server}, c, r), \mathsf{Response}, \mathsf{Error})$ to S and abort.
        Let $k = \mathtt{getKey}(s, (\mathsf{Server}, c, r))$.
        Let $m_s = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)$.
        Let $\sigma_s = \mathtt{sign}(s, (\mathsf{Server}, c, r), m_s)$.
        Update $(t, r, c)$ to $(t, r, \varepsilon)$ in $L[s]$.
        Send $(m_s, \sigma_s)$ to $A_{\mathrm{S}}$.
```

```
Deliver a response to the client:
processClientReceive($m_s, \sigma_s$)
    Decode $m_s$ into ($\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s$).
    If $state[c, s, r] \neq \mathsf{OK}$, $cor[\mathsf{Client}, c, (\mathsf{Client}, s, r)]$, or $|p_s| > n[c, s, r]$, abort.
    Let $n[c, s, r] = n[c, s, r] - |p_s|$, abort.
    Let $k = \mathtt{getKey}(s, (\mathsf{Server}, c, r))$.
    Let $b = \mathtt{verify}(s, (\mathsf{Server}, c, r), \mathsf{Server}, m_s, \sigma_s, k)$.
    If $b \neq 1$, abort.
    Let $state[c, s, r] = \mathsf{Stopped}$.
    Send ($s$, ($\mathsf{Server}, c, r$), $\mathsf{Response}, \mathsf{Send}$) to S.
Reset of the server:
processServerReset($s$)
    If $state[s] \neq \mathsf{OK}$ or $cor$, abort.
    Let $state[s] = \mathsf{Reset}$.
    Let $t = \mathtt{getTime}(s, \mathsf{Server})$.
    If $t \geq t[s]$, let $t[s] = t$.
    Let $state[s] = \mathsf{OK}$, $t^{\min}[s] = t[s] + \mathrm{tol}^{+}[s]$ and $R[s] = L[s] = [\,]$.
Corrupt a machine and if necessary, note which one was corrupted:
processCorruptionRequest($id_1, ..., id_n, T$)
    If $T = A_{\mathrm{S}}$, let $cor[\mathsf{Server}, id_1] = \mathsf{true}$.
    If $T = A_{\mathrm{C}}$, let $cor[\mathsf{Client}, id_1, id_2] = \mathsf{true}$.
    If $T = A_{\mathrm{sig}}$, let $cor[\mathrm{Sig}, id_1, id_2] = \mathsf{true}$.
    If $T = A_{\mathrm{ver}}$, let $cor[\mathrm{Sig}, id_1, id_2, id_3] = \mathsf{true}$.
    Send ($id_1, ..., id_n, \mathsf{Corrupt}$) to C, S, or Sig.
Corrupt a Server:
corruptServer($pid$)
    Let $cor[\mathsf{Server}, s] = \mathsf{true}$.
    Send ($pid$, ($\mathsf{Server}$), $\mathsf{Corrupt}$) to S
    Receive ($pid$, ($\mathsf{Server}$), $\mathsf{Corrupted}, x$) from S.
Get the time of a principal:
getTime($pid, sid$)
    Send ($pid, sid, \mathsf{GetTime}$) to LC.
    Recv ($pid, sid, \mathsf{Time}, t$) from LC.
    Return $t$.
Get a key from the keystore:
getKey($pid, sid$)
    Send ($pid, sid, \mathsf{GetKey}$) to KS.
    Recv ($pid, sid, \mathsf{PublicKey}, k$) from KS.
    Return $k$.
Get a signature:
sign($pid, sid, m$)
    If $cor[\mathrm{Sig}, pid, sid]$, abort.
    Send ($pid, sid, \mathsf{Sign}, m$) to Sig.
    Recv ($pid, sid, \mathsf{Signature}, \sigma$) from Sig.
    Return $\sigma$.
Verify a signature:
verify($pid, sid, ssid, m, \sigma, k$)
    Send ($pid, sid, ssid, \mathsf{Init}$) to Sig.
    Recv ($pid, sid, ssid, \mathsf{Init}$) from Sig.
    If $cor[\mathrm{Sig}, pid, sid, ssid]$, abort.
    Send ($pid, sid, ssid, \mathsf{Verify}, m, \sigma, k$) to Sig.
    Recv ($pid, sid, ssid, \mathsf{Verified}, b$) from Sig.
    Return $b$.
```