INSTITUT FÜR INFORMATIK

Computationally Secure Two-Round Authenticated Message Exchange

Klaas Ole Kürtz
Henning Schnoor
Thomas Wilke

Bericht Nr. 0810
May 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Computationally Secure Two-Round Authenticated Message Exchange

Klaas Ole Kürtz
Henning Schnoor
Thomas Wilke

Technical Report Nr. 0810
May 2009

{kuertz|schnoor|wilke}@ti.informatik.uni-kiel.de

This is an updated version (May 2009)
of the original Technical Report Nr. 0810 (September 2008).

This work was supported by the DFG under grant KU 1434/4-1.

Abstract. We study two-round authenticated message exchange protocols consisting of a single request and a single response, with the realistic assumption that the responder is long-lived and has bounded memory. We first argue that such protocols necessarily need elements such as timestamps to be secure. We then present such a protocol and prove that it is correct and computationally secure.
In our model, the adversary provides the initiator and the responder with the payload of their messages, which means our protocol can be used to implement securely any service based on authenticated message exchange. We even allow the adversary to read and reset the memory of the principals and to use, with very few restrictions, the private keys of the principals for signing the payloads or parts thereof. The latter corresponds to situations in which the keys are not only used by our protocol. We use timestamps to secure our protocol, but only assume that each principal has access to a local clock.

This is an updated version (May 2009) of the original Technical Report Nr. 0810 (September 2008). While the overall structure of the definitions and results in this paper remained the same, some details have been changed; we point out the main differences in Appendix A.

1 Introduction

A characteristic feature of web services (see, e. g., [ML07,LB07]) and other services provided in the Internet (such as remote procedure call [Sun98,Win99]) is their restricted form of communication. Unlike in other cryptographic settings, these protocols have only two rounds: In the first round, a client sends a single message (request) to a server; in the second round, the server replies with a single message (response) containing the result of processing the request. A central security goal arising is that of authenticated message exchange: The server wants to be convinced that the request is new and originated from the alleged client, while the client wants to be convinced that the response originated from the intended server and is a response to his request.

The main objective of this paper is to provide a security model for such *two-round authenticated message exchange protocols (2AMEX protocols)* and to prove that a natural and practical protocol, which follows recommendations stated in various places of the literature, see, for instance, [NKMHB06], is computationally secure, assuming that the underlying signature scheme is resistant against existential forgeries.

A simple protocol to achieve the aforementioned security goals works as follows: The client appends a message id (e. g., random nonce or sequence number) to his actual request, signs the result, and sends the signed message to the server. The server verifies the signature on the received message and checks that it has not seen the message id previously. It takes the result of processing the request, appends the message id he received from the client, signs the result, and sends the signed message to the client. Finally, the client verifies the signature and the message id.— The problem here is that the server needs to keep track of all message id's it has seen, because otherwise it is easy to mount replay attacks. In this paper, we show how to use timestamps in order to ensure security even with bounded memory.

The fact that we want to devise a single protocol for implementing arbitrary services brings up the following issue. In contrast to what happens in traditional authentication protocols, say in authenticated key exchange (see, e. g., [BR93]), where the messages exchanged have a fixed format, we have to allow the messages exchanged to carry a so-called *payload*, which can be of arbitrary and to the authentication protocol unknown structure. In particular, we want to allow that a payload contains security-related data such as signed parts, as defined in [NKMHB06]. As a realistic assumption, the private key that a principal uses for such signatures inside the payload may be the same as the one used for signing entire protocol messages. But obviously, in order for the entire protocol to be secure such a use of private keys to sign parts of the payload has to be restricted. To account for this, our protocol and security model is such that, on the one hand, the adversary is allowed to generate all payloads, but, on the other hand, he is only provided with a protocol-dependent signature oracle (rather than all private keys).

We analyze our protocol in a model adapted from [BR93]. Another option would be to use a simulation-based model such as Canetti's Universal Composition model [Can01], Küster's model using inexhaustible Turing machines [Küs06] or Backes, Pfitzmann, and Waidner's Cryptographic Library [BPW03]. These models have the feature that they provide a notion of composition which allows for modular security proofs. However, there does not seem to be a way to "decompose" the long-lived server algorithm in our protocol into simpler components such that these models simplify our security proof.

Related work. To the best of our knowledge there is no prior work on the security of authenticated message exchange in a setting similar to the one we study. In contrast, there is a wide range of papers on entity authentication protocols (often in connection with key exchange), see, for instance, the seminal paper on entity authentication and key exchange, [BR93], which our paper is based on. Bellare and Rogaway's paper has a very brief section about "authenticated exchange of text", which discusses how in a three-round entity authentication protocol authenticated data can be transmitted. In that paper, the authors do not, however, give a formal definition of authenticated exchange of text nor do they consider two-round protocols nor is their setting general enough to support an arbitrary service using this protocol. Entity authentication has also been studied in the Universal Composition model [CH06] and in combination with the cryptographic library [BP03]; a computational analysis of the Needham-Schroeder-Lowe entity authentication protocol [NS78,Low96] is given in [War05]. Another crucial difference to our model is that in the mentioned papers, the responder (server) is short-lived, whereas in our model a server processes an unbounded number of requests from different clients, which is reminiscent of optimistic contract-signing protocols, see [ASW98,GJM99], where the trusted third party potentially needs to remember an unbounded number of requests.

Timestamps, which are crucial to our work, have been used in various cryptographic settings, for instance, in a key exchange protocol proposed in [DS81]. In [DG04,BEL05] symbolic models for protocols with timestamps are introduced and techniques to analyze protocols within these models are described. In [KLP07] the timing model is similar to ours, however, the paper is concerned with secure multi-party computation.

In our model, we allow the adversary to reset the server at any time; in [BFGM01] resetting of principals is discussed in a different context.

2 The Protocol 2AMEX-1—Informal Description

In this section, we describe our protocol *2AMEX-1* informally.

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

1. a) c is asked by a user to send the request p_c
   b) $c \rightarrow s\colon \{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c}$
   c) s checks whether the message is admissible and if not, stops
   d) s forwards the request (r, p_c) to the addressed service
2. a) s receives a response (r, p_s) from the service
   b) s checks whether the response is admissible and if not, stops
   c) $s \rightarrow c\colon \{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)\}_{sk_s}$
   d) c forwards the response p_s to the user

Here, r is a randomly chosen message identity which is also used as a handle by the (see steps 1. d) and 2. a)), t is the local time 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 $t_{\min}$ and a finite set 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 denotes the local time of the server, and L is set to the empty set.

Step 1. c). Upon receiving a message as above, the server rejects if $t \notin \left[t_{\min} + 1, t_s + \mathrm{tol}_s^+\right]$ or if $(t', r, c') \in L$ for some t' and c', and otherwise proceeds as follows: If L contains less than cap_s elements, it inserts (t, r, c) into L. If L contains cap_s elements or more, the server deletes all tuples containing the oldest timestamp from L, until L contains less than cap_s tuples. Then it sets $t_{\min}$ 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.

In the rest of this paper we give a formal framework for specifying and analyzing such protocols, in particular, we define what it means for such protocols to be correct and secure, and prove that 2AMEX-1 is indeed correct and secure, provided the underlying signature scheme is so.

3 Protocol Model

The formal framework we are working in is an adaptation of the framework for entity authentication introduced by Bellare and Rogaway in [BR93] to the message authentication setting.

As mentioned earlier, in a bounded memory setting time is necessary to achieve resistance against replay attacks. We use l_{time}-bit numbers as time values for an arbitrary fixed $l_{\mathrm{time}} \in \mathbb{N}$. We also assume there is an arbitrary fixed *identifier set* $\mathrm{IDs} \subseteq \{0,1\}^{l_{\mathrm{ID}}}$ for an arbitrary fixed $l_{\mathrm{ID}} \in \mathbb{N}$ whose elements are called *identifiers*. We use them to identify principals, which can act both as clients and as servers.

Restricting IDs to be a finite set and fixing l_{time} makes the presentation more accessible. All of our results remain true when these parameters are varied with the security parameter: The length of time values l_{time} and the length of identifiers l_{ID} could grow polynomially with the security parameter.

3.1 Signature Schemes

Our message exchange protocols use signature schemes, where a *signature scheme* $\mathbf{S}$ is a triple of algorithms (G, S, V), satisfying the following conditions:

- G is a *key generation algorithm*, i. e., a probabilistic polynomial-time algorithm which on input 1^k, with k being the security parameter, produces a pair (pk, sk), where pk is a public key and sk the corresponding secret key,
- S is a *signing algorithm*, i. e., a probabilistic polynomial-time algorithm which for any bit string $b \in \{0,1\}^*$ and any secret key sk produces a signature $S(b, sk)$, and
- V is a deterministic *verification algorithm* which on input $((b, S(b, sk)), pk)$ returns true if (pk, sk) has been generated by G.

By $\{b\}_{sk}$, we denote the pair $(b, S(b, sk))$, i. e., the bit string b accompanied by a valid signature obtained from the signature scheme. In the remainder of the paper, we use a fixed signature scheme.

3.2 Clients and Servers

Before defining clients and servers formally, we describe how they are supposed to operate. An intended run of an authenticated message exchange protocol between a client $c \in \mathrm{IDs}$ and a server $s \in \mathrm{IDs}$ is initiated by the client-side environment which wants to call some service on the server. The protocol run consists of two rounds, request and response, modeled by four steps as illustrated in Figure 1 (cf. our protocol description in Section 2):

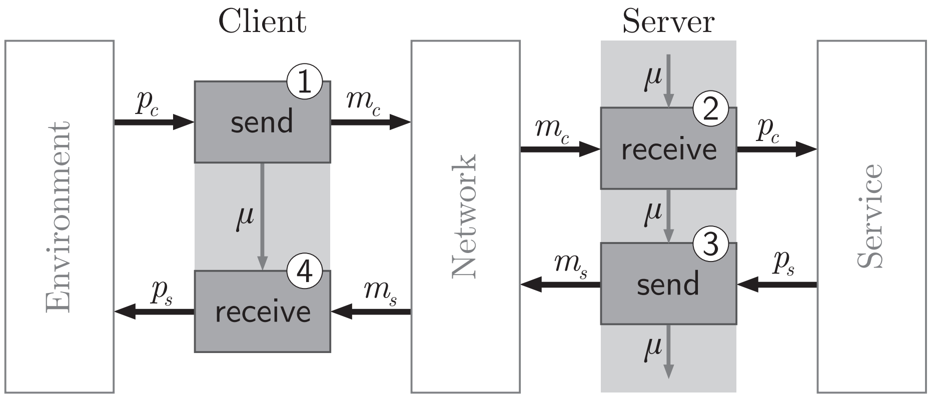

Fig. 1. Message flow in four steps.

client send The client is given a request payload p_c by the environment which is a request to the service provided by the server s. The client encapsulates the payload, adding security data etc., and sends the resulting message m_c over the network.

server receive The server receives the message m_c from the network, accepts the message and unwraps it, giving the payload p_c, a handle h, and the identified sender of the incoming message c to the service.

server send The server is provided with a response payload p_s and the handle h by the service (which chose p_s as a response to the request payload p_c). The server encapsulates the payload and sends a message, m_s, over the network.

client receive Finally, the client receives the message m_s from the network and returns p_s to the environment.

To give the strongest security guarantees possible, the roles of the environment, the service, and the network are all played by the adversary in our security model. As the adversary is free to choose any payload, our protocols support any service.

This leads to the following formal definitions. A *client algorithm* is a probabilistic polynomial-time algorithm. As first input parameter, the client gets an instruction which can either be send or receive. The same is true for a *server algorithm* with the only difference that for a server algorithm there is a third instruction, reset. Table 1

input	client Γ	server Σ
instruction	$\alpha \in \{\mathsf{send}, \mathsf{receive}\}$	$\alpha \in \{\mathsf{send}, \mathsf{receive}, \mathsf{reset}\}$
security parameter	1^k	1^k
identity	$c \in \text{IDs}$	$s \in \text{IDs}$
partner's identity	$s \in \text{IDs}$	
public keys	pk_{IDs}	pk_{IDs}
private key	sk_c	sk_s
local time	$t \in \{0,1\}^{l_{\text{time}}}$	$t \in \{0,1\}^{l_{\text{time}}}$
payload or message	p or $m \in \{0,1\}^*$	p or $m \in \{0,1\}^*$
message handle		$h \in \{0,1\}^*$
state information	μ	μ

output	client Γ	server Σ
message or payload	m or $p \in \{0,1\}^*$	m or $p \in \{0,1\}^*$
decision	$\delta \in \{\mathsf{A}, \mathsf{R}\}$	$\delta \in \{\mathsf{A}, \mathsf{R}\}$
assumed partner		$c \in \text{IDs} \cup \{\varepsilon\}$
message handle		$h \in \{0,1\}^*$
state information	μ'	μ'

Table 1. Input and output values of the algorithms Γ and Σ.

specifies the input parameters and output values of client and server algorithms. We first explain the input parameters for the client; then we turn to the server.

Client and Server Algorithms: Input and Output. First of all, the client algorithm gets a *security parameter* which is used to define polynomial running time. Second, the client algorithm is provided with the identifier of the principal it is running for, $c \in \text{IDs}$, and with the identifier of the server it is supposed to be talking to, $s \in \text{IDs}$. Third, the client algorithm is provided with the family of public keys, $pk_{\text{IDs}} = \{pk_a\}_{a \in \text{IDs}}$, and its own private key sk_c. Fourth, it gets the local time $t \in \{0,1\}^{l_{\text{time}}}$. Fifth, the client is provided with the payload $p \in \{0,1\}^*$ it is supposed to send to the server, or with a message $m \in \{0,1\}^*$ obtained from the network. Finally, our client processes multiple requests, one after the other, which means is has a history or, in other words, a state. We model this by state information that it is provided with—the parameter $\mu \in \{0,1\}^*$, initialized with ε.

For the server instructions, the situation is similar. But a server can receive input from various clients, so it is not provided with a particular client identifier. Rather, the server has to extract this from the message it receives and store it to send a response later on. When asked to respond to a specific message, the server is also provided by the service with a message handle identifying the message the service wants to respond to. Also, for a reset instruction, the state information and the message given to the server is ε.

Next, we explain the output values for the server. First of all, when receiving a message m_c, the server algorithm extracts the payload $p \in \{0,1\}^*$ carried by m_c and returns it. Second, the server algorithm reports his *decision* $\delta \in \{\mathsf{A}, \mathsf{R}\}$: A (*accept*) means that the command was executed successfully, while R (*reject*) indicates an error (which can be a failed authentication or another protocol error). Third, the server algorithm reports the identifier of the client it assumes it received a message from or it intends to send a message to. When the decision is R, the dummy value $\varepsilon$is used. Fourth, the server outputs a message handle $h \in \{0,1\}^*$. When the service wants to respond to m_c, it has to provide the server with this exact message handle. Finally, the server outputs state information, which it will be provided with the next time it will be called, unless it is reset.

Clients have the same output syntax except that there is no need to output a message handle or the assumed partner, because the latter is contained in the input values of the algorithm.

Execution Orders. There are only certain sequences of instructions to client and server algorithms that make sense: We require the client to (i) only accept the first send request it receives, (ii) accept at most one $\mathsf{receive}$ request, and (iii) accept a $\mathsf{receive}$ request only after it accepted a send request. The server is required to accept a send request with message handle h if there is a previous $\mathsf{receive}$ request it accepted earlier with the same message handle h, and if between these both requests it accepted no other request.

This can be formalized as follows, where we start with the client. Let $k \in \mathbb{N}$, let c, s be identifiers, let $\mu_0 = \varepsilon$, let $\{\alpha_j\}_{j\in\mathbb{N}}$ be a sequence of instructions with $\alpha_j \in \{\mathsf{send}, \mathsf{receive}\}$, let $\{t_j\}_{j\in\mathbb{N}}$ be an monotonically increasing sequence of timestamps and $\{b_j\}_{j\in\mathbb{N}}$ a sequence of bit strings. Assume that for all $i \in \mathbb{N}$ we have

$$\Gamma(\alpha_i, 1^k, c, s, pk_{\mathrm{IDs}}, sk_c, t_i, b_i, \mu_i) = (b'_i, \delta_i, \mu_{i+1}) \enspace ,$$

then we require that (i) only for the smallest $i_1 \in \mathbb{N}$ with $\alpha_{i_1} = \textsf{send}$ we have $\delta_{i_1} = \textsf{A}$, if such an i_1 exists; (ii) there is at most one $i_2 \in \mathbb{N}$ with $\alpha_{i_2} = \textsf{receive}$ and $\delta_{i_2} = \textsf{A}$; and (iii) if there is i_2 as in (ii), then there is an i_1 as in (i) with $i_1 < i_2$.

For the server, let k, s, $\{t_j\}_{j\in\mathbb{N}}$ and $\{b_j\}_{j\in\mathbb{N}}$ be as above, let $\{h_j\}_{j\in\mathbb{N}}$ be a sequence of message handles, and let $\{\alpha_j\}_{j\in\mathbb{N}}$ be a sequence of instructions with $\alpha_j \in \{\textsf{send}, \textsf{receive}, \textsf{reset}\}$ and $\mu'_{-1} = \varepsilon$. If for all $i \in \mathbb{N}$ we have

$$\Sigma(\alpha_i, 1^k, s, pk_{\mathrm{IDs}}, sk_s, t_i, b_i, h_i, \mu_i) = (b'_i, \delta_i, c_i, h'_i, \mu'_i)$$
$$\text{with } \mu_i = \begin{cases} \varepsilon & \text{if } \alpha_i = \textsf{reset} \\ \mu'_{i-1} & \text{otherwise,} \end{cases}$$

then we require that for each pair $i_1, i_3 \in \mathbb{N}$ with $i_1 < i_3$, $\alpha_{i_1} = \textsf{receive}$, $\delta_{i_1} = \textsf{A}$, $\alpha_{i_3} = \textsf{send}$, and $h'_{i_1} = h_{i_3}$, that $\delta_{i_3} = \textsf{A}$ if there is no $i_2 \in \mathbb{N}$ with $i_1 < i_2 < i_3$ and $\delta_{i_2} = \textsf{A}$.

3.3 2AMEX Protocols, Adversary, and the Experiment

We now give the formal definition of a *Two-Round Authenticated Message Exchange (2AMEX) protocol*. Such a protocol is a tuple $\Pi = (\Gamma, \Sigma, \tau, F, E)$ where

- Γ is a client algorithm,
- Σ is a server algorithm,
- τ is a *time function* (see below),
- F is a *freshness function* (also see below), and
- E is an *exception set* as defined below.

A *time function* is a function that assigns to each client message m_c a time value $\tau(m_c)$. The intended interpretation is that $\tau(m_c)$ is the time at which m_c was created. The time function well be used to phrase the correctness condition (see Section 5).

A *freshness function* is a function which, for an identity s, state information μ_s and a time t_s, specifies a freshness interval $F(s, \mu_s, t_s)$, see Section 4 for an example. This is the interval of time values the server s considers fresh, i. e., for the server to consider a message fresh the time value of that message has to be in the server's freshness interval.

An *exception set* is a set of bit strings called *exceptions* which is recognizable in polynomial time. This is the set of bit strings which the signature oracle (see below) will refuse to sign for the adversary.

The Experiment. We now describe how all these components work together. This is done, as usual, by defining a notion of experiment, which also incorporates the notion of adversary.

For every principal $a \in \mathrm{IDs}$ a *server instance* Σ_a runs under the identity a. For every principal $c \in \mathrm{IDs}$ and every principal $s \in \mathrm{IDs}$ arbitrarily many *client instances* $\Gamma^i_{c,s}$ can run where c acts as a client and s as a server, and where i is a natural number.

We let the adversary control all these instances, that is, the adversary can decide when to call such an instance, which payloads to choose, which local times are used, etc. The only restriction that we impose is that local times are *monotone*, i. e., for each principal, the value of the local clock in each step of the experiment cannot be smaller than the value in the previous step.

There is a *signature oracle*, denoted with $\mathcal{S}$, which can be used by the adversary (i) to sign bit strings while he constructs the payload for a send instruction, and (ii) to corrupt a principal's key. The corresponding instructions are sign and corrupt. Clearly, we cannot allow the adversary to use the signature oracle to sign every bit string. Therefore, the exception set refuses to sign bit strings belonging to the exception set specified in the protocol description.

The experiment works in *steps*, where in each step the adversary can choose local times for the involved principals and perform an action (send, receive, reset, sign, corrupt), for which he provides the parameters under his control and receives the output values. The details of this process are given in Table 2. Note that the adversary even has access to the local state information μ of the principals. This implies that a principal cannot store its secret key locally without compromising the security of the protocol—however this is not necessary anyway, since the secret key is given to the client and server algorithms in each step.

In the experiment *traces* are recorded for each instance, which allow us to define correctness and security of a protocol, see the next section. A trace is a sequence of tuples containing a step number and the observable action of the instance in the corresponding step, i. e., the local time t, the payloads and messages received or sent by the instance in this step, as well as the decision of the instance (accept or reject), and finally, for servers entries denoting the identity of the client that the server believes it is communicating with and the message handle.

The experiment $\mathsf{Exp}_{\Pi,\mathcal{A},k}$ for an adversary $\mathcal{A}$ against a protocol Π as above with security parameter k proceeds as described in Table 2, where we use $v \xleftarrow{R} A$ to describe assigning the output of the (randomized) algorithm A to the variable v.

4 The Protocol 2AMEX-1—Formal Description

In this section, we recast our protocol 2AMEX-1 within the formal framework developed in the previous section, and comment on various aspects of it.

4.1 Formal Description

Recall the informal description from Section 2 and the formal appearance of a 2AMEX protocol from Section 3. What we have to specify is a signature scheme, the server algorithm as well as the client algorithm, the time function as well as the freshness function, and the exceptions set.

Signature scheme. For the signature scheme, we allow an arbitrary one that is resistant against existential forgery (see Section 6.2 for details).

Server Algorithm, Freshness Function, and Time Function. Let s be the identity that the server algorithm Σ is called with. As state information μ, the server uses a tuple $(t_{\min}, L)$ consisting of a variable $t_{\min}$ holding a single timestamp and a set L of triples of the form (t, r, c) where t is a timestamp, r is a message id, and c is an identity.

The freshness function is defined by $F(s, (t_{\min}, L), t) = \left[t_{\min} + 1, t + \mathrm{tol}_s^+\right]$.

The server first checks if it is called with state information ε and if so (i. e. initially and after each reset), sets $t_{\min}$ to $t_s + \mathrm{tol}_s^+$ where t_s is the current local time of the server, and sets L to the empty set. Then the server proceeds according to the instruction.

1. *Generate keys.*
   For each $a \in \mathrm{IDs}$:
   (a) Let $(pk_a, sk_a) \overset{R}{\longleftarrow} G(1^k)$.
   (b) Send (a, pk_a) to the adversary.
2. *Initialize clocks.*
   For each $a \in \mathrm{IDs}$, let $t_a(0) \longleftarrow 0$.
3. *Initialization of the clients.*
   For each $i \in \mathbb{N}$ and $c, s \in \mathrm{IDs}$, let $\mathrm{tr}^i_{c,s} \longleftarrow \varepsilon$ and $\mu^i_{c,s} \longleftarrow \varepsilon$.
4. *Initialize states and traces of the servers.*
   For each $s \in \mathrm{IDs}$, let $\mu_s \longleftarrow \varepsilon$ and $\mathrm{tr}_s \longleftarrow \varepsilon$.
5. *Initialize step counter.*
   Let $n \longleftarrow 0$.
6. *Run the adversary.*
   Run the adversary $\mathcal{A}$ step by step and do the following for each step:
   a) Let $n \longleftarrow n + 1$.
   b) For each $a \in \mathrm{IDs}$, let $\mathcal{A}$ choose the value $t_a(n) \in \{0,1\}^{l_{\mathrm{time}}}$ with $t_a(n) \geq t_a(n-1)$.
   c) Call client, server or signature algorithm as follows according to the adversary's selection:
      – $\Gamma^i_{c,s}$: $\mathsf{send}(p)$
        (i) $(m, \delta, \mu) \overset{R}{\longleftarrow} \Gamma(\mathsf{send}, 1^k, c, s, pk_{\mathrm{IDs}}, sk_c, t_c(n), p, \mu^i_{c,s})$,
        (ii) $\mu^i_{c,s} \longleftarrow \mu$,
        (iii) $\mathrm{tr}^i_{c,s} \longleftarrow \mathrm{tr}^i_{c,s} \cdot (n, \mathsf{send}, t_c(n), p, m, \delta)$,
        (iv) return (m, δ, μ) to the adversary.
      – Σ_s: $\mathsf{receive}(m)$
        (i) $(p, \delta, c, h, \mu) \overset{R}{\longleftarrow} \Sigma(\mathsf{receive}, 1^k, s, pk_{\mathrm{IDs}}, sk_s, t_s(n), m, \varepsilon, \mu_s)$,
        (ii) $\mu_s \longleftarrow \mu$,
        (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{receive}, t_s(n), p, m, \delta, c, h)$,
        (iv) return (p, δ, c, h, μ) to the adversary.
      – Σ_s: $\mathsf{send}(p, h)$
        (i) $(m, \delta, c, h', \mu) \overset{R}{\longleftarrow} \Sigma(\mathsf{send}, 1^k, s, pk_{\mathrm{IDs}}, sk_s, t_s(n), p, h, \mu_s)$,
        (ii) $\mu_s \longleftarrow \mu$,
        (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{send}, t_s(n), p, m, \delta, c, h)$,
        (iv) return (m, δ, c, h', μ) to the adversary.
      – $\Gamma^i_{c,s}$: $\mathsf{receive}(m)$
        (i) $(p, \delta, \mu) \overset{R}{\longleftarrow} \Gamma(\mathsf{receive}, 1^k, c, s, pk_{\mathrm{IDs}}, sk_c, t_c(n), m, \mu^i_{c,s})$,
        (ii) $\mu^i_{c,s} \longleftarrow \mu$,
        (iii) $\mathrm{tr}^i_{c,s} \longleftarrow \mathrm{tr}^i_{c,s} \cdot (n, \mathsf{receive}, t_c(n), p, m, \delta)$,
        (iv) return (p, δ, μ) to the adversary.
      – Σ_s: $\mathsf{reset}()$
        (i) $(m, \delta, c, h, \mu) \overset{R}{\longleftarrow} \Sigma(\mathsf{reset}, 1^k, s, pk_{\mathrm{IDs}}, sk_s, t_s(n), \varepsilon, \varepsilon, \varepsilon)$,
        (ii) $\mu_s \longleftarrow \mu$,
        (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{reset}, t_s(n), \varepsilon, \varepsilon, \mathsf{A}, \varepsilon, \varepsilon)$,
        (iv) return (m, δ, c, h, μ) to the adversary.
      – $\mathcal{S}$: $\mathsf{corrupt}\,(a)$
        (i) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{corrupt}, t_s(n), \varepsilon, \varepsilon, \mathsf{A}, \varepsilon, \varepsilon)$,
        (ii) return sk_a to the adversary.
      – $\mathcal{S}$: $\mathsf{sign}(a, p)$
        (i) If $p \notin E$, return $\{p\}_{sk_a}$ to the adversary, otherwise return ε to the adversary.

Table 2. The experiment $\mathsf{Exp}_{\Pi,\mathcal{A},k}$ for an adversary $\mathcal{A}$ against a protocol $\Pi = (\Gamma, \Sigma, \tau, F, E)$ with security parameter k.

Upon receiving $m_c = \{(\mathsf{From}\colon c, \mathsf{To}\colon s', \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c}$, at local server time t_s with state information $\mu = (t_{\min}, L)$, the server s performs the following:

1. If one of the following conditions is met, stop and return $(\varepsilon, \mathsf{R}, \varepsilon, \varepsilon, \mu)$:
   (a) $s' \neq s$,
   (b) $V(m_c, pk_c)$ returns *false*,
   (c) $t \notin F(s, \mu, t_s)$,
   (d) $(t', r, c') \in L$ for some t', c'.
2. While $|L| \geq \mathrm{cap}_s$,
   (a) $t_{\min} \longleftarrow \min\{t' \mid (t', r', c') \in L\}$,
   (b) $L \longleftarrow \{(t', r', c') \in L \mid t' > t_{\min}\}$.
3. $L \longleftarrow L \cup \{(t, r, c)\}$.
4. Return $(p_c, \mathsf{A}, c, r, (t_{\min}, L))$.

Observe that this corresponds to steps 1. c) and 1. d) of the informal description in Section 2.

The following corresponds to steps 2. a)–c) of the informal description in Section 2. When asked to send a payload p_s with message handle r and state information $\mu = (t_{\min}, L)$, the server algorithm proceeds as follows:

1. Look for $(t, r, c) \in L$ with $c \neq \varepsilon$. If no matching triple is found in the list, return $(\varepsilon, \mathsf{R}, \varepsilon, \varepsilon, \mu)$.
2. $m_s \longleftarrow \{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)\}_{sk_s}$.
3. $L \longleftarrow (L \setminus \{(t, r, c)\}) \cup \{(t, r, \varepsilon)\}$.
4. Return $(m_s, \mathsf{A}, c, \varepsilon, (t_{\min}, L))$.

The time function is defined by $\tau(m_c) = t$ where m_c is as above.

Client Algorithm. Let c be the client identity that Γ is called with. If the instruction is to send a payload p_c to server s at time t and the state information μ is ε, the algorithm randomly chooses the message id $r \xleftarrow{R} \{0,1\}^k$, sets $m_c = \{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c}$ and returns (m_c, A, r). If requested to send when $\mu \neq \varepsilon$, it returns $(\varepsilon, \mathsf{R}, \mu)$. Note that this corresponds to steps 1. a) and 1. b) of the informal description in Section 2.

The following corresponds to steps 2. c) and 2. d) of the informal description in Section 2. If the algorithm is instructed to receive a message $m_s = \{(\mathsf{From}\colon s', \mathsf{To}\colon c', \mathsf{Ref}\colon r', \mathsf{Body}\colon p'_s)\}_{sk_{s'}}$ when the state information is μ, it proceeds as follows:

1. If one of the following conditions is met, stop and return $(\varepsilon, \mathsf{R}, \mu)$:
   (a) $|\mu| \neq k$,
   (b) $s' \neq s$,
   (c) $c' \neq c$,
   (d) $V(m_s, pk_s)$ returns *false*,
   (e) $r' \neq \mu$.
2. Return $(p_s, \mathsf{A}, 0^{k+1})$.

Bit String Representations and Exceptions. Our description above leaves open the actual format of the messages. We assume that the tags ($\mathsf{From}, \dots$) and tuples which form the messages are represented as bit strings in such a way that the individual components can be retrieved without ambiguity.

The set $E \subseteq \{0,1\}^*$ is the set of all bit string representations of messages of the form $(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)$ or $(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)$. We assume the bit string representation is such that E is recognizable in polynomial time. For example, by using SOAP [ML07] one can meet these requirements.

This completes the definition of our protocol. Note that it can easily be seen that the restrictions on execution orders from Section 3.2 hold. Also, it is easy to see that our protocol indeed achieves to work with bounded memory as desired:

Remark 4.1. The size of the state of a server s is bounded by the size of the bit string representation of $(t_{\min}, L)$, where $t_{\min} \in \{0,1\}^{l_{\text{time}}}$ is a timestamp and L is a list of cap_s many tuples of the form (t, r, c) with $t \in \{0,1\}^{l_{\text{time}}}$, $r \in \{0,1\}^k$ and $c \in \{0,1\}^{l_{\text{ID}}}$.

4.2 Comments and Caveats

Resets. From the specification of 2AMEX-1, it is immediate that after a reset there is a delay in accepted messages: If a reset of a server s happens at a step n_r, then the next accepted message must have a timestamp exceeding $t_s(n_r) + \text{tol}^+_s$. However, such a delay is natural, since for any protocol that resists replay attacks, if a reset happens at step n_r, and $n_1 < n_r < n_2$, then the intervals $F(s, \mu_s(n_1), t_s(n_1))$ and

$F(s, \mu_s(n_2), t_s(n_2))$ must be disjoint. Due to asynchronous clocks, we need the interval $F(s, \mu_s(n), t_s(n))$ to exceed the time $t_s(n)$, therefore rejecting valid messages cannot be completely avoided.

To illustrate this, assume that a protocol is designed in such a way that immediately after a reset, i. e., without an increase in the server time, the interval of accepted messages is not empty, and there is a message m that the server accepts. Then the adversary can simply reset the server, deliver the message m, and then reset and deliver again, without ever changing the value of the server clock. Since for the server, the two events of receiving the message m are indistinguishable, it accepts the message twice. Therefore, in any secure protocol, the interval F will be empty when a reset happened, as long as the clock of s has not been increased. It easily follows from inspection of our protocol (as well as from the above reasoning and our later security proof) that in 2AMEX-1 this is the case.

Parameterization. Our protocol is parameterized, since tol_s^+ and cap_s can be chosen freely. We will see that for any choice of tol_s^+ and cap_s the protocol is correct and secure—however, our correctness definition relies on "reasonable" values for the intervals F. A message m sent by a client c in step n_1 and received by a server s in step n_2 is rejected if $t_c(n_1) = \tau(m) \notin F(s, \mu_s(n_2), t_s(n_2))$. By construction of the protocol, there are two ways in which this can happen: (i) $t_c(n_1) > t_s(n_2) + \mathrm{tol}_s^+$, or (ii) $t_c(n_1) \leq t'_{\min}$ where $t'_{\min}$ is s's internal variable $t_{\min}$ before step n_2.

The first of these issues can occur when the clocks of client and server are asynchronous, which in realistic environments is very likely. To circumvent this problem, one should choose the constant tol_s^+ large enough to deal with usually occurring time differences between the local clocks of the principals.

The second case occurs after a reset or if, in step n_2, the server s has accepted more messages with timestamps in the future of $t_c(n_1)$ than the maximal number of message id entries it can maintain in the set L. This can happen, for instance, due to network properties that slow down the delivery of messages. Obviously, increasing cap_s makes this case occur less frequently, in particular, if the servers would have unbounded memory, it would not occur at all.

Responding to old Messages. A protocol is only required to allow the service to respond to the most recently received and accepted message (see Execution Orders in Section 3.2). But a good protocol should allow the service to respond to more, i. e. older messages, while still accepting incoming messages. In our protocol, we can give the following guarantee on how long the service will be able to respond to a message:

Let t be a timestamp and let $\mu = (t_{\min}, L)$ be the state information of a server s. Assume that L already contains n_1 tuples whose timestamps are older than t, and let $n_2 = \mathrm{cap}_s - |L|$. Now if a message m is received and accepted with $\tau(m) > t_{\min}$, the service will be able to respond to m using its message handle as long as the server, after accepting m, does not accept more than $n_1 + n_2$ messages with a timestamp greater than or equal to $\tau(m)$.

Dishonest Timestamps. In a way, the protocol 2AMEX-1 gives the clients incentive to "lie" in their timestamps, since for the clients, it is advantageous to claim a timestamp in the future, as long as the timestamp does not exceed the sum of the server clock plus its tolerance. Assume, for example, that the server tolerance tol_s^+ is very large, let's say 24 hours. Then a client has an advantage if it adds 24 hours to the timestamp of each message that it sends to the server s, since its messages will most likely not be rejected due to old timestamps. This has an unwanted effect on the operation of the server: If this client (or a group of clients acting in the same way) sends many requests to the server, and if the server does not have enough memory, the value $t_{\min}$ of s will soon be in the future as well, which leads to the rejection of valid incoming messages. The consequence of this line of thought is that in practice, it is desirable that the "center" of the intervals F should always be the present time, so that the most successful strategy for the clients is to use truthful timestamps.

In Section 7, we explain how this can be achieved.

5 Correctness and Security Definitions

We now define what it means that a protocol is correct and secure in our model. For a fixed execution of the experiment, an identifier

s and a natural number n, we define $\mu_s(n)$ to be the content of the state information μ_s *before* the nth step. We say that for a principal $a \in \text{IDs}$ the principal's key is *corrupted* in the experiment at step n, if there is a step number $n' \leq n$ such that in step n', the adversary performed a $\mathcal{S}\colon \mathsf{corrupt}\,(a)$ query.

From now on, with $\text{tr}^i_{c,s}$ and tr_s, we refer to the corresponding traces *after* running the experiment.

5.1 Correctness Definition

Informally, our notion of correctness requires that if messages are delivered as intended by the network (i. e., the adversary), then all parties accept (given that the messages are considered fresh by the servers), the sender of each message is correctly determined, and the payloads are delivered correctly. Formally, we say that an adversary $\mathcal{A}$ is *benign* if it only delivers messages that were obtained from a client or server instance, and delivers a message at most once to every instance. This models a situation in which arbitrary payload is sent over a network in which messages may get lost, all messages can be read by anybody, and servers can loose state information, but no message is altered, no false messages are introduced, and no replay attacks are attempted.

Definition 5.1. *A 2AMEX protocol Π is* correct *if it satisfies the following for any benign adversary $\mathcal{A}$, any security parameter k, and any $c, s \in \text{IDs}$:*

1. *If $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A}) \in \text{tr}^i_{c,s}$, $(n_2, \mathsf{receive}, t_2, p'_c, m_c, \delta_s, c', h) \in \text{tr}_s$, and $\tau(m_c) \in F(s, \mu_s(n_2), t_2)$, then $c' = c$, $p_c = p'_c$, and $\delta_s = \mathsf{A}$, with all but negligible probability.*
2. *If additionally $(n_3, \mathsf{send}, t_3, p_s, m_s, \mathsf{A}, c', h) \in \text{tr}_s$ and $(n_4, \mathsf{receive}, t_4, p'_s, m_s, \delta_c) \in \text{tr}^i_{c,s}$ with $n_2 < n_3$ and $n_1 < n_4$, but with no $(n', \ldots, \mathsf{A}, \ldots) \in \text{tr}_s$ having $n_2 < n' < n_3$, then we also require that $p_s = p'_s$ and $\delta_c = \mathsf{A}$.*

Note that this definition leaves a loop hole for "correct", but utterly useless protocols: The freshness function F is part of the specification, and a protocol only has to be correct with regard to this choice of F. Hence a protocol in which F always returns the empty interval is not required to accept any messages. For protocols to be useful in practice, it is desirable to have a large freshness interval.

Similarly, this definition only guarantees that the service can respond to the last message that the server received and accepted. Using message handles, a good protocol should allow the service to respond to any of the recently received messages.

The reason why we only require the server to accept with all but negligible probability is that we allow randomness in our algorithms, and therefore collisions cannot be ruled out completely.

5.2 Security Definition

We will now define when a protocol is called secure by defining a function which matches client and server traces. We will only consider the *acceptance trace* of a client instance $\Gamma^i_{c,s}$, which is the subsequence of all steps in the trace $\mathrm{tr}^i_{c,s}$ of the form $(n, \ldots, \mathsf{A})$. We also say that an instance *accepts at step* n if there is an entry of the form $(n, \ldots, \mathsf{A})$ or $(n, \ldots, \mathsf{A}, \ldots)$ in its trace.

Depending on the result of the experiment, we define the event $\mathsf{NoMatching}_{\Pi,\mathcal{A},k}$, which is intended to model the event that the adversary $\mathcal{A}$ has successfully "broken" the protocol Π with security parameter k. A *partner function* is a partial map $f \colon \mathrm{IDs} \times \mathrm{IDs} \times \mathbb{N} \dashrightarrow \mathbb{N}$. Informally, for each client instance $\Gamma^i_{c,s}$, the function f points to a step (identified by step counter n) in which the server accepts the message sent from c to s in session i, if there is such a step.

If a "matching" partner function (see below) can be defined, then the experiment was successful in the sense that the adversary did not compromise authenticity of the message exchange. More formally, matching w. r. t. a given partner function is defined as follows.

1. A trace $\mathrm{tr}^i_{c,s}$ of a client c *matches the server trace* tr_s of the server s w. r. t. a given partner function f if the acceptance trace of $\Gamma^i_{c,s}$ is of the form $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A})(n_4, \mathsf{receive}, t_4, p_s, m_s, \mathsf{A})$ and there are timestamps t_2, t_3, step numbers $n_1 < n_2 < n_3 < n_4$, and a handle h such that $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h) \in \mathrm{tr}_s$ and $(n_3, \mathsf{send}, t_3, p_s, m_s, \mathsf{A}, c, h) \in \mathrm{tr}_s$, and $f(c, s, i) = n_2$.
2. A step $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h)$ in the trace tr_s of a server s *matches the client trace* $\mathrm{tr}^i_{c,s}$ of the client c w. r. t. a given partner

function f if $f(c,s,i) = n_2$ and the first accepting step in $\mathrm{tr}^i_{c,s}$ is of the form $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A})$ for some t_1 and $n_1 < n_2$.

For a partner function f, the event $\mathsf{NoMatching}^f_{\Pi,\mathcal{A},k}$ (designed to model that f is not a partner function that validates the communication in the result of the experiment) consists of two cases:

(a) There are parties c and s, a session number i, and a step number n_4, such that c and s are not corrupted at step n_4, the client instance $\Gamma^i_{c,s}$ accepts at step n_4, but the trace $\mathrm{tr}^i_{c,s}$ does not match the server trace tr_s w. r. t. f, or

(b) there are parties c and s and a step number n_2, such that c is not corrupted at step n_2, and there is a step $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h) \in \mathrm{tr}_s$ which does not match the client trace $\mathrm{tr}^i_{c,s}$ w. r. t. f for any session number i.

The event $\mathsf{NoMatching}_{\Pi,\mathcal{A},k}$ denotes the event that $\mathsf{NoMatching}^f_{\Pi,\mathcal{A},k}$ occurs for *all* partial functions $f\colon \mathrm{IDs} \times \mathrm{IDs} \times \mathbb{N} \dashrightarrow \mathbb{N}$ when the experiment is run with protocol Π, adversary $\mathcal{A}$ and security parameter k, i. e., the event that there does not exist a partner function that validates the success of the experiment.

The *advantage of an adversary* $\mathcal{A}$ running against Π is the probability that the adversary is successful in breaking the protocol, formally defined by:

$$\mathsf{Adv}_{\Pi,\mathcal{A}}(k) = \Pr\left[\mathsf{NoMatching}_{\Pi,\mathcal{A},k}\right].$$

Definition 5.2. *A 2AMEX protocol Π is* secure *if for every polynomial-time adversary $\mathcal{A}$ the advantage of $\mathcal{A}$, $\mathsf{Adv}_{\Pi,\mathcal{A}}(k)$, is a negligible function of the security parameter k.*

Note that this notion of security also rules out replay attacks: If a server accepts a message m_c twice from the same client c, then in the trace tr_s there are two *different* entries $(n_1, \mathsf{receive}, t_1, p^1_c, m_c, \mathsf{A}, c, h^1)$ and $(n_2, \mathsf{receive}, t_2, p^2_c, m_c, \mathsf{A}, c, h^2)$, where $n_1 \neq n_2$. If the event $\mathsf{NoMatching}$ does not occur, then, by definition, there must be a partner function f and tuples (c,s,i_1) and (c,s,i_2) such that $f(c,s,i_1) = n_1$ and $f(c,s,i_2) = n_2$. Since f is a function and $n_1 \neq n_2$, it follows that $i_1 \neq i_2$. Therefore, the client c did send the message m_c twice: once in session i_1, and once in session i_2.

Hence, our notion of security does allow a server to accept the same message twice, but only if it also has been sent twice. However, since there is no communication between server and client except for the exchanged messages, the server has no way of knowing whether a message that has been received twice was also sent twice. Therefore, protocols satisfying our security definition will have to be designed in such a way that a message is accepted at most once by a server (with all but negligible probability).

Note that it is of course allowed for the server to accept the same *payload* twice from the same client.

6 Correctness and Security Proof

6.1 Correctness of 2AMEX-1

Theorem 6.1. *2AMEX-1 is a correct 2AMEX protocol.*

Proof. Assume that $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A}) \in \mathrm{tr}^i_{c,s}$ and $(n_2, \mathsf{receive}, t_2, p'_c, m_c, \delta_s, c', h) \in \mathrm{tr}_s$ in an experiment where $\mathcal{A}$ is a benign adversary, and assume that $t_1 \in F(s, \mu_s(n_2), t_2)$.

First, note that the probability that the message id of m_c is the same as a message that was previously delivered is negligible, since $\mathcal{A}$ is benign and therefore delivers m_c at most once to s, and the probability of a message id being generated twice is negligible. Hence, we can assume $n_1 < n_2$. We show that the probability of all four cases that lead to rejection of the message on the server side is negligible. Since m_c was created by the client instance $\Gamma^i_{c,s}$, we know that the To- and From-fields of m_c are s and c, respectively, and that m_c was signed with c's private key. Due to the above, we also know that the probability of m_c's message id already appearing in the set L maintained by s is negligible. Finally, the message cannot be rejected in step $1(c)$, since by the prerequisites, $\tau(m_c) = t_1 \in F(s, \mu_s(n_s), t_2)$. Thus, the server accepts with all but negligible probability. By construction of the protocol, it is also clear that the server concludes that the message has been sent by c, and that $p'_c = p_c$ because the Body-Field of m_c equals p_c.

Now assume that additionally $(n_3, \mathsf{send}, t_3, p_s, m_s, \mathsf{A}, c', h) \in \mathrm{tr}_s$ and $(n_4, \mathsf{receive}, t_4, p'_s, m_s, \delta_c) \in \mathrm{tr}^i_{c,s}$ with $n_2 < n_3$ and $n_1 < n_4$, but with no $(n', \ldots, \mathsf{A}, \ldots) \in \mathrm{tr}_s$ having $n_2 < n' < n_3$.

First, we know that the server only generates one response for the incoming message m_c (as he overwrites c with ε in the tuple (t, r, c) in L after sending the response), and since the adversary is benign, this response is delivered only once to c, so n_4 is the only step in which a response can be accepted by c. Now we know that the probability of rejection by the client is zero, because the To-field of the response is set to c, the message id is correct as it was stored in the server's memory (which was not reset between n_2 and n_3), and the server's signature is correct. Thus, the client accepts the message at n_4 and we also have $p'_s = p_s$ because the Body-field of the response is set to p_s by the server. □

In the remainder of this section we show that the protocol 2AMEX-1 satisfies our security definition, provided the signature scheme used satisfies a standard security requirement.

We define the following notation: For a server identity s, let $t^s_{\min}(n)$ denote the value of s's internal variable $t_{\min}$ before step n.

6.2 Security of Signature Schemes

Our security proof for 2AMEX-1 will rely on the signature scheme being resistant against *existential forgery.* We briefly explain this notion: An adversary against a signature scheme is a probabilistic polynomial-time algorithm which as input receives the security parameter 1^k, a public key pk generated by the key generation algorithm, and has access to a signature oracle $\mathcal{O}$, which on input m generates a valid signature of m corresponding to the public key pk. The signature scheme is *resistant against existential forgery* if any adversary against the scheme has negligible probability (in the security parameter) of producing a pair (m, s), where s is a valid signature of m (corresponding to the key pk), and s has not been obtained by querying the oracle $\mathcal{O}$.

We adopt this definition to our scenario, which requires handling of signatures of a group of users as well as dynamic corruption. An *adaptively corrupting n-user adversary* against the signature scheme $\mathbf{S}$ is a probabilistic polynomial-time algorithm $\mathcal{A}$ that gets as input n public keys generated by the key generation algorithm of $\mathbf{S}$, and can perform the following actions:

– Obtain, via a signature oracle, a valid signature for any message and any of the involved public keys,
– via a corrupt (i)-query, obtain the secret key corresponding to the ith public key.

The experiment in which such an adversary is run proceeds as follows: First, n public/private key pairs $(pk_1, sk_1), \ldots, (pk_n, sk_n)$ are generated by the key generation algorithm of $\mathbf{S}$, the public keys are handed to the adversary. Then the adversary is run, its queries are answered accordingly. The adversary is successful if it can generate a triple (m, s, i), such that no corrupt (i)-query was performed by the adversary before, the signature s was not obtained by an oracle query, and s is a valid signature for m for the public key pk_i. In this case we also say that the adversary has achieved existential forgery against pk_i (without prior corruption).

We say that a signature scheme $\mathbf{S}$ is *secure in the n-user setting with adaptive corruption* if every adversary has negligible success probability (in the security parameter).

It is obvious that a signature scheme meeting the above security definition is also resistant against existential forgery. We now prove that the converse is also true:

Lemma 6.2. *Let* $\mathbf{S}$ *be a signature scheme that is resistant against existential forgery, and let n be a fixed natural number. Then* $\mathbf{S}$ *is also secure in the n-user setting with adaptive corruption.*

Proof. Assume that there is an adversary $\mathcal{A}_{\text{SIG}}^{\text{adapt}}$ against $\mathbf{S}$ that has non-negligible success probability in the n-user setting with adaptive corruption. We construct an adversary $\mathcal{A}_{\text{SIG}}^{\text{single}}$ that has non-negligible probability of achieving existential forgery against $\mathbf{S}$ in the usual sense.

$\mathcal{A}_{\text{SIG}}^{\text{single}}$ gets a single public key pk_x (generated by the key generation algorithm of $\mathcal{S}$) and a signature oracle $\mathcal{O}$ for this key. It chooses $i \leq n$ randomly and sets $pk_i = pk_x$. Using the key generation algorithm, it generates $n - 1$ public/secret key pairs (pk_j, sk_j) for $j \leq n, j \neq i$, and simulates the adversary $\mathcal{A}_{\text{SIG}}^{\text{adapt}}$, handling oracle queries regarding the key pk_i by querying the oracle $\mathcal{O}$, and oracle queries regarding the public key pk_j for $j \neq i$ by computing the signature itself with the secret key sk_j. Queries of the form corrupt (j)

are handled by returning sk_j if $j \neq i$, if a query $\mathsf{corrupt}\,(i)$ is made, $\mathcal{A}_{\text{SIG}}^{\text{single}}$ reports failure.

Since pk_i is generated with the same key generation algorithm as the other keys, the probability that the simulated $\mathcal{A}_{\text{SIG}}^{\text{adapt}}$ obtains a valid signature for pk_i (without a $\mathsf{corrupt}\,(i)$-query preceding the success) is the same as that it achieves existential forgery (without preceding corruption) w. r. t. any of the other keys. In particular, since the number of keys is fixed, the probability of achieving existential forgery without prior corruption against the key pk_i is non-negligible. It is clear that in the cases where $\mathcal{A}_{\text{SIG}}^{\text{adapt}}$ achieves existential forgery for pk_i, $\mathcal{A}_{\text{SIG}}^{\text{single}}$ is successful. Therefore, $\mathcal{A}_{\text{SIG}}^{\text{single}}$ has a non-negligible success probability. This is a contradiction, since S is secure against existential forgery. □

We also note that it has been observed that resistance against existential forgery is not sufficient to achieve natural security goals in the multi-user setting [MS04], however this level of security suffices to obtain security in our protocol.

6.3 Security of 2AMEX-1

Theorem 6.3. *2AMEX-1 is a secure 2AMEX protocol, provided that the signature scheme used resists existential forgery.*

We first show that 2AMEX-1 is resistant against replay attacks. The following lemma states that the same message is not accepted twice by a server during a protocol run:

Lemma 6.4. *Let $\mathcal{A}$ be an adversary, k a security parameter, and $s \in$ IDs. Then in a run of $\mathsf{Exp}_{2AMEX\text{-}1,\mathcal{A},k}$, if $(n_1, \mathsf{receive}, t_s(n_1), p_1, m_1, \mathsf{A}, c_1, h_1)$ and $(n_2, \mathsf{receive}, t_s(n_2), p_2, m_2, \mathsf{A}, c_2, h_2)$ are entries in tr_s with $m_1 = m_2$, then $n_1 = n_2$.*

Proof. Assume that a server s accepts a message $m = \{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon x)\}_{sk_c}$ twice, at steps n_1 and n_2, where $n_1 < n_2$. Then at the step n_1, the pair (t, r, c) is inserted into L. At point n_2, since s accepts the message m again, we know that (t, r, c) is not contained in L anymore. Also, $t^s_{\text{min}}(n_2) < t$ (otherwise, s rejects).

Assume there was no reset between n_1 and n_2. Since (t, r, c) has been removed from L at some point before n_2, we know that $t^s_{\min}(n_2) \geq t$ due to the construction of the protocol. This is a contradiction to the above.

Hence a reset happened at step n_r, where $n_1 < n_r < n_2$. Due to the monotonicity of the clocks, $t_s(n_1) \leq t_s(n_r)$. Since the server accepted the message m with timestamp t at point n_1, we know that $t \leq t_s(n_1) + \mathrm{tol}^+_s$. We also know that $t_s(n_r) + \mathrm{tol}^+_s \leq t^s_{\min}(n_2)$, since the server runs 2AMEX-1. Therefore we conclude $t^s_{\min}(n_2) < t \leq t_s(n_1) + \mathrm{tol}^+_s \leq t_s(n_r) + \mathrm{tol}^+_s \leq t^s_{\min}(n_2)$—a contradiction. $\square$

Note that the preceding proof of Lemma 6.4 is the only situation where we actually use monotonicity of the clocks—it is clear that clocks are needed only to circumvent replay attacks. Also, it is immediate from the proof that it suffices to demand that clocks of participants who act in the server role are monotone. We now can prove Theorem 6.3:

Proof. Assume there is a polynomial-time adversary $\mathcal{A}_{\mathrm{AUT}}$ that achieves non-negligible probability of NoMatching. We construct an adaptively corrupting |IDs|-user adversary against the signature scheme **S** used by 2AMEX-1 that has non-negligible success probability. Due to Lemma 6.2 it then follows that the signature scheme is not resistant against existential forgery, which is a contradiction. Note that in the protocol 2AMEX-1, the adversary can compute the local state information μ of all involved principals by only observing their output behavior. We therefore can assume that that the adversary disregards the state information given to it by the experiment. In particular, no security relevant information is given to the adversary after the initialization phase, except the messages and payloads sent and received by the principals, the fact whether they accept or reject incoming requests, the identified communication partner when a server instance is called, and secret keys obtained with $\mathcal{S}$: corrupt (a) queries.

We construct the adversary $\mathcal{A}_{\mathrm{SIG}}$ against the signature scheme. By definition, $\mathcal{A}_{\mathrm{SIG}}$ has access to signature oracles and public keys for all involved parties, as well as access to private keys pk_i via corrupt (i)-queries. $\mathcal{A}_{\mathrm{SIG}}$ proceeds as follows: It initializes the variables of the experiment (traces, private information, etc.) as in a legitimate run of $\mathsf{Exp}_{\text{2AMEX-1},\mathcal{A}_{\text{AUT}},k}$. It then simulates $\mathcal{A}_{\text{AUT}}$, handling calls of instances as follows: When a client or server instance is called by $\mathcal{A}_{\text{AUT}}$, then $\mathcal{A}_{\text{SIG}}$ simulates the client or server algorithm, where it derives the input parameters for the client or server algorithm (except for the secret key) in the exact same way as defined in the experiment: From $\mathcal{A}_{\text{AUT}}$, $\mathcal{A}_{\text{SIG}}$ obtains a message or a payload, and generates the remaining arguments depending on the type of action defined by the adversary call (send, $\mathsf{receive}$, or reset). During this simulation, $\mathcal{A}_{\text{SIG}}$ uses the signature oracles to generate the signatures that are signed by client and server algorithms for messages that are elements of E, i. e., are unsigned 2AMEX-1 messages. Note that every message which was signed in this way appears in the trace of the corresponding principal. Queries of the form $\mathcal{S}\colon \mathsf{corrupt}\,(a)$ are answered by using the $\mathsf{corrupt}\,(i)$-queries available to $\mathcal{A}_{\text{SIG}}$. Finally, when the simulated $\mathcal{A}_{\text{AUT}}$ uses a query of the form $\mathcal{S}\colon \mathsf{sign}(a,p)$, the adversary $\mathcal{A}_{\text{SIG}}$ uses the signature oracle to obtain the corresponding signature if $p \notin E$, and returns ε otherwise. Note for later reference that this ensures that every signature for a valid 2AMEX-1 message that $\mathcal{A}_{\text{AUT}}$ did not generate internally (possibly with access to the secret key after corruption) appears in the trace of the corresponding principals.

Additionally, $\mathcal{A}_{\text{SIG}}$ keeps track of a step counter for the number of steps, as well as the traces and internal variables of server and client oracles as in the original experiment. Finally, $\mathcal{A}_{\text{SIG}}$ constructs a partner function as follows: For every client instance $\varGamma^i_{c,s}$, if the first accepting step in $\mathrm{tr}^i_{c,s}$ (which must be a send-instruction) is $(n, \mathsf{send}, t, p, m, \mathsf{A})$, then let $f(c,s,i) = n'$, where n' is the smallest step number referring to an accepting $\mathsf{receive}$-query of the server instance $\varSigma_s$ with incoming message m, if such a step exists. Let $f(c,s,i)$ be undefined otherwise. Since valid 2AMEX-1 messages can be recognized in polynomial time, and $\mathcal{A}_{\text{AUT}}$ runs in polynomial time, $\mathcal{A}_{\text{SIG}}$ runs in polynomial time. By construction, $\mathcal{A}_{\text{SIG}}$ is an adaptively corrupting $|\text{IDs}|$-user adversary against S. We show that $\mathcal{A}_{\text{SIG}}$ has non-negligible success probability.

Since for the simulated $\mathcal{A}_{\text{AUT}}$, there is no difference between this experiment and the original $\mathsf{Exp}_{\text{2AMEX-1},\mathcal{A}_{\text{AUT}},k}$, the newly constructed adversary $\mathcal{A}_{\text{SIG}}$ has non-negligible probability of achieving $\mathsf{NoMatching}$ defined with respect to the simulated experiment in the obvious way. We now show that unless message id's for different messages coincide, each occurrence of $\mathsf{NoMatching}$ w. r. t. f implies existential forgery in the multi-user setting against a public key that has not been corrupted. Since $\mathcal{A}_{\mathrm{SIG}}$ can verify whether a message it produces contains a signature that is a successful forgery, every event of $\mathsf{NoMatching}$ thus allows $\mathcal{A}_{\mathrm{SIG}}$ to produce a forged signature. Therefore, if $\mathcal{A}_{\mathrm{AUT}}$ is successful in achieving $\mathsf{NoMatching}$ with non-negligible probability, then $\mathcal{A}_{\mathrm{SIG}}$ is successful in achieving existential forgery with non-negligible probability as required (note that collision of message id's happens with negligible probability only).

It remains to prove the above claim: Every occurrence of $\mathsf{NoMatching}$ implies a collision of message id's, or existential forgery against the signature scheme. We show this indirectly: Assume that the event $\mathsf{NoMatching}$ occurs, and neither existential forgery against an uncorrupted key, nor collision of message id's occurs. In particular, $\mathsf{NoMatching}^f$ appears, where f is the partner function constructed above. We distinguish the two cases in the definition of $\mathsf{NoMatching}^f$ (see Section 5.2).

First Case. Assume that case (a) occurs, and existential forgery and the collision of message id's both did not occur. By definition of the $\mathsf{NoMatching}$ event, there are parties c, s, a session number i, and a step n_4 such that c and s are not corrupted at step n_4, the client $\Gamma^i_{c,s}$ accepted at n_4, but $\mathrm{tr}^i_{c,s}$ does not match the server trace tr_s w. r. t. f. This means that the accepting steps of $\mathrm{tr}^i_{c,s}$ are of the form $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A})(n_4, \mathsf{receive}, t_4, p_s, m_s, \mathsf{A})$, but there are no t_2, t_3, n_2, n_3, h' with $n_1 < n_2 < n_3 < n_4$, such that $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h') \in \mathrm{tr}_s$ and $(n_3, \mathsf{send}, t_3, p_s, m_s, \mathsf{A}, c, h') \in \mathrm{tr}_s$ with $f(c, s, i) = n_2$. Since both c and s are not corrupt at step n_4, the signature oracle available to $\mathcal{A}_{\mathrm{AUT}}$ does not allow the signing of valid protocol messages, and we assumed that existential forgery did not occur, it follows that every valid protocol message signed with the keys of c or s that was obtained before the step n_4 were obtained by a call of the client or server instance.

Since the client $\Gamma^i_{c,s}$ accepted the incoming message m_s, we know that m_s is a valid 2AMEX-1 message send by a server with s's signature. Note that 2AMEX-1 allows to distinguish messages sent by client or by servers: The former contain a message id, the latter a reference to one. By the above, this means that $\mathcal{A}_{\mathrm{AUT}}$ obtained m_s from a call to the server instance Σ_s. By construction of the protocol, this means that there is an entry $(n_3, \mathsf{send}, t_3, p'_s, m_s, \mathsf{A}, c', h)$ in the server trace tr_s, and since $\mathcal{A}_{\mathrm{AUT}}$ had access to m_s in step n_4, it follows that $n_3 < n_4$. Since the client instance $\Gamma^i_{c,s}$ extracted the payload p_s from m_s, and the server instance Σ_s encapsulated the payload p'_s into m_s, it follows that $p_s = p'_s$. Since $\Gamma^i_{c,s}$ accepts m_s, it is addressed to c, and by construction of the protocol it follows that $c = c'$. Therefore the above step in tr_s is $(n_3, \mathsf{send}, t_3, p_s, m_s, \mathsf{A}, c, h)$, with $n_3 < n_4$.

Further, we know that a server s accepts a send-request only if there is a preceding receive-request accepted by s with a matching message handle (i. e., a message id). Hence there is an entry $(n_2, \mathsf{receive}, t_2, p'_c, m'_c, \mathsf{A}, c'', h)$ in the trace tr_s with $n_2 < n_3$, and there is no accepted receive instruction or send instruction with message handle h in tr_s with a step number between n_2 and n_3. By construction of the protocol, it follows that $c'' = c$. Since Σ_s accepts the message m'_c and determines the sender to be $c'' = c$, it follows that m'_c is a valid 2AMEX-1 client message, is addressed to s, and carries a correct signature for c's key. Due to the above, and since m'_c is addressed to the server s, we can assume that m'_c was obtained by the call of a client instance $\Gamma^{i'}_{c,s}$. Hence there is an entry $(n'_1, \mathsf{send}, t'_1, p''_c, m'_c, \mathsf{A})$ with $n'_1 < n_2$ in the client trace $\mathrm{tr}^{i'}_{c,s}$. Since the payload p''_c was encapsulated into m'_c, and p'_c was extracted from m'_c, it follows that $p''_c = p'_c$.

Since $\Gamma^i_{c,s}$ accepts m_s, we know that (due to the verification of message id's, and since we assumed that collision of id's does not occur) m_s contains a reference to the message id of m_c, which encapsulated the payload p_c. Since m_s was created by Σ_s using the message handle that Σ_s output when processing m'_c, we know from the construction of 2AMEX-1 that m_s carries a reference to the message id contained in m'_c. Hence m_c and m'_c have the same message id, and by the above assumption it follows that $m'_c = m_c$, implying $p'_c = p_c = p''_c$. It follows that the above step in tr_s is of the form $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h)$. Again due to our assumption that collisions of message id's do not occur, and since m_c was created in both the client session i and in the session i', it further follows that $i = i'$

and thus $n_1 = n'_1$, which implies $n_1 < n_2 < n_3 < n_4$. In particular, the message m_c was sent by the client instance $\Gamma^i_{c,s}$.

We now show that $f(c, s, i) = n_2$. By construction, since m_c is the message created by the client instance $\Gamma^i_{c,s}$, $f(c, s, i) = n$, where n is the lowest step number such that Σ_s accepted the message m_c in step n. By the above, we know that Σ_s accepted m_c in step n_2. By Lemma 6.4, we know that a server accepts a message at most once. Hence it follows that $n_2 = n$, and by the steps exhibited in the server trace tr_s above, we know that the trace $\mathrm{tr}^i_{c,s}$ matches the server trace tr_s w. r. t. f—a contradiction.

Second Case. In case (b), there are parties c and s and a step n_2 such that c is not corrupted in step n_2, and there is a step $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h)$ in the trace tr_s which does not match $\mathrm{tr}^i_{c,s}$ for any session number i, i. e., there is no i such that the first accepting entry in $\mathrm{tr}^i_{c,s}$ is of the form $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A})$ for some $n_1 < n_2$ such that $f(c, s, i) = n_2$.

Since s accepts m_c and determines that it has been sent by c, we know that m_c carries a valid signature by c, and is a 2AMEX-1 message. Since we assume that existential forgery does not occur, c is not corrupt in step n_2, and m_c is addressed to s, we know that m_c was obtained from a client instance $\Gamma^i_{c,s}$. Hence there is an entry $(n_1, \mathsf{send}, t_1, p'_c, m_c, \mathsf{A})$ in $\mathrm{tr}^i_{c,s}$, with $n_1 < n_2$ (since m_c must be obtained before the adversary can use it). Since p'_c is the payload encapsulated in m_c and p_c is the payload extracted from p_c, it follows that $p_c = p'_c$. Hence the above step is of the form $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A})$. Since m_c is the message created by the instance $\Gamma^i_{c,s}$ and m_c was accepted by Σ_s in step n_2 (and, by Lemma 6.4, in no other step), it follows that $f(c, s, i) = n_2$. Hence the step $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A})$ matches the trace $\mathrm{tr}^i_{c,s}$—a contradiction. □

7 Practical Considerations and "good" parameters

A weakness of the protocol as stated in Section 4 are the rather vague guarantees implied by our security definition. As discussed at the end of Section 4.2, a certain type of denial-of-service attack can be mounted against the protocol, which results in the intervals F

being empty, or to be in the future entirely, essentially rendering a server inaccessible for all parties who set their clocks honestly.

Therefore, as mentioned before, it is important to choose the parameters for the server, i. e., the tolerance tol_s^+ and the capacity cap_s in a way that circumvents problems like this. In the following theorem, we specify one way of choosing values for these parameters, that imply "liveliness" of the servers at all times.

Theorem 7.1. *Let s be a server running 2AMEX-1, and let tol_s^- be a real number such that*

- *the minimal time (measured by the server's local clock) between accepting two messages as well as between a reset and accepting the first message is at least t_{diff},*
- *the server tolerance is tol_s^+,*
- $\mathrm{cap}_s > \dfrac{(\mathrm{tol}_s^+ + \mathrm{tol}_s^-)}{t_{\mathrm{diff}}}$.

Then for any local server time t_s, if the last reset (or initialization) of s happened before $t_s - (\mathrm{tol}_s^+ + \mathrm{tol}_s^-)$, then $t^s_{\min} \leq t_s - \mathrm{tol}_s^-$.

Before proving the theorem, we explain its significance. The claim that it it establishes is that (resets aside), the value $t_{\min}$ is always at least tol_s^- units of time before the current server time. Hence tol_s^- is the minimal amount of time that the server can "look into the past" via its recorded set of messages, and by the way that the protocol is designed, this means that messages with a timestamp set this much in the past (relative to the local server time) can still get accepted. Hence the value tol_s^- is a "backwards tolerance" with respect to out-of-sync clocks in the same way as tol_s^+ gives "forward tolerance". For practical choices of these values, one should keep in mind that tol_s^- also needs to compensate for the network delay between sending and receiving a message, hence arguably "backward tolerance" should be higher than "forward tolerance".

The reason why the theorem only guarantees the inequality for the case that at least tol_s^- units of time have passed since the last reset is that as discussed in Section 4.2, after a reset, there must be a time where no incoming message can be accepted, and obviously one has to wait longer to ensure that messages with timestamps further in the past can be accepted again.

We now prove the theorem.

Proof. Assume the last reset (or initialization, which for the server is the same event) of s happened at the time t^s_r (measured in the clock of s). Fix a sequence of incoming messages since the last reset. We obviously are only interested in accepted messages, since rejected messages do not lead to an advance of the value t_{min}. Further, assuming that all messages in the sequence are accepted, we are not interested in the messages themselves or even the sender and message id's, but only in the time at which they are received by s, and the timestamp they carry. Hence we consider a sequence of messages as a sequence of pairs $M = (t^c_i, t^s_i)_{i\in\mathbb{N}}$, where a pair (t^c_i, t^s_i) represents a message that the server s receives at time t^s_i, and which carries the client's timestamp t^c_i. Since the minimal delay between incoming messages and between a reset and an incoming message is t_{diff}, we require that $t^s_r + t_{\text{diff}} \leq t^s_0$, and $t^s_i + t_{\text{diff}} \leq t^s_{i+1}$ for all i. We also require that $t^c_i \leq t^s_i + \text{tol}^+_s$ for all i (other sequences cannot be accepted by the server). With $t^s_{\text{min}}(M)(t^s)$ we denote the value of t_{min} at the local server time t^s, when the server s receives the sequence M (obviously, for this value only the elements in M with an incoming time of at most t_s are considered).

It is easy to see that $t^s_{\text{min}}(M)(t^s)$ for a fixed t^s, considered as a function in M, is *monotone* in the following sense: Lowering an incoming-time value of a pair or increasing the timestamp of a pair in M does not decrease the value of $t^s_{\text{min}}(M)(t^s)$, as long as the modified sequence still obeys the restrictions explained above. It therefore follows that we only have to consider the extreme case where messages come with the highest possible frequency and having the highest (at that time) admissible timestamp, i. e., we only need to consider the *canonical sequence* $M_c = (t^s_r + i \cdot t_{\text{diff}}, t^s_r + \text{tol}^+_s + i \cdot t_{\text{diff}})_{i\geq 1}$. This sequence M_c can be thought of as the optimal denial of service attack against the server s. By construction of the protocol and due to choice of cap_s, s only removes elements from L if there are more than $(\text{tol}^+_s + \text{tol}^-_s)/t_{\text{diff}}$ elements in the set L.

The claim that we need to prove is:

$$\text{if } t \geq t^s_r + \text{tol}^+_s + \text{tol}^-_s \text{ then } t^s_{\text{min}}(M_c)(t) \leq t - \text{tol}^-_s.$$

We first consider the case $t = t_r^s + \mathrm{tol}_s^+ + \mathrm{tol}_s^-$. In this case, exactly $\mathrm{tol}_s^+ + \mathrm{tol}_s^-$ units of time have passed since the last reset. In this time, s has accepted exactly ${(\mathrm{tol}_s^+ + \mathrm{tol}_s^-)}/{t_{\mathrm{diff}}}$ messages, which is less than cap_s. Therefore, no element has been removed from the set, and $t_{\min}$ still has the value that it was set to at the last reset, which is $t_r^s + \mathrm{tol}_s^+$ by the specification of the protocol. Hence $t_{\min}^s(M_c)(t) = t_r^s + \mathrm{tol}_s^+ = t - \mathrm{tol}_s^-$, which proves the required inequality. For points in time beyond $t = t_r^s + \mathrm{tol}_s^+ + \mathrm{tol}_s^-$, it suffices to prove that $t_{\min}$ does not advance faster than t^s. This is easy to see, since by the setup of the sequence M_c, $t_{\min}$ advances by exactly t_{diff} for each element removed from the set L, and for each received message, at most one message is removed from this set (since all messages have different timestamps). Finally, the delay between the acceptance of two messages, and hence the minimal delay between advancements of $t_{\min}$, is exactly t_{diff}. Therefore, given the sequence M_c, the value $t_{\min}$ increases at most as fast as the server clock, and hence the inequality is maintained. $\square$

8 Conclusion

For the the natural security goal of authenticated message exchange in two rounds, we introduced a computational model and security definition. We also presented a protocol 2AMEX-1 which uses timestamps to achieve security even with bounded memory. We proved security in our model even if the adversary is allowed to reset the memory of long-lived servers and has partial access to the signature scheme. This models the realistic situation that our protocol does not have exclusive access to the public-key infrastructure.

References

ASW98. N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures (extended abstract). In *EUROCRYPT*, pages 591–606, 1998.

BEL05. Liana Bozga, Cristian Ene, and Yassine Lakhnech. A symbolic decision procedure for cryptographic protocols with time stamps. *Journal of Logic and Algebraic Programming*, 65(1):1–35, 2005.

BFGM01. Mihir Bellare, Marc Fischlin, Shafi Goldwasser, and Silvio Micali. Identification protocols secure against reset attacks. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 495–511. Springer, 2001.

BP03. Michael Backes and Birgit Pfitzmann. A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS*, pages 1–12, 2003.

BPW03. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 220–230. ACM, 2003.

BR93. M. Bellare and P. 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. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 136–145. IEEE Computer Society, 2001.

CH06. Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 380–403. Springer, 2006.

DG04. Giorgio Delzanno and Pierre Ganty. Automatic verification of time sensitive cryptographic protocols. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2004.

DS81. Dorothy E. Denning and Giovanni M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

GJM99. Juan A. Garay, Markus Jakobsson, and Philip D. MacKenzie. Abuse-free optimistic contract signing. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 449–466. Springer, 1999.

KLP07. Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. Concurrent composition of secure protocols in the timing model. *J. Cryptology*, 20(4):431–492, 2007.

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`.

Low96. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

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/`.

MS04. Alfred Menezes and Nigel Smart. Security of signature schemes in a multi-user setting. *Designs, Codes and Cryptography*, 33(3):261–274, 2004.

NKMHB06. Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: SOAP message security 1.1 (WS-Security 2004). Technical report, OASIS Web Services Security TC, 2006. OASIS Standard.

NS78. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

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

War05. Bogdan Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. *Journal of Computer Security*, 13(3):565–591, 2005.

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

A Differences to the Previous Version of this Work

A.1 Changes in the May 2009 Version from the September 2008 Version

In our model, we introduced *handles* (see Section 3.2) to allow the server to respond not only to the most recently received message, but to more messages. This resulted in several small changes in the model, the correctness and security definitions, the protocol, and the correctness and security proofs. The main change are:

– The client and server algorithms now have input and output parameters for handles, see Section 3.2.
– The definition of execution orders in Section 3.2 has been adjusted such that it guarantees that the service is at least able to respond to the most recently received message.
– To be consistent with this definition, we switched steps 2 and 3 in the server algorithm in Section 4 and adjusted some details accordingly.