Computationally Secure Two-Round Authenticated Message Exchange [⋆]

Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke

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

Abstract. We prove secure a concrete and practical two-round authenticated message exchange protocol which reflects the authentication mechanisms for web services discussed in various standardization documents. The protocol consists of a single client request and a subsequent server response and works under the realistic assumptions that the responding server is long-lived, has bounded memory, and may be reset occasionally. The protocol is generic in the sense that it can be used to implement securely any service based on authenticated message exchange, because request and response can carry arbitrary payloads. The security analysis we provide is a computational analysis in the Bellare-Rogaway style and thus provides strong guarantees; it is novel from a technical point of view since we extend the Bellare-Rogaway framework by timestamps and payloads with signed parts.

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.

Protocols which are as described above are called *two-round authenticated message exchange protocol (2AMEX protocol)* in this paper. The underlying authentication issues and mechanisms for dealing with them are discussed in various papers, see, for instance, [BFGP03,BFG04], and also dealt with in practice, see, for instance, [NKMHB06]. However, no concrete 2AMEX protocol has been formally specified, let alone rigorously analyzed with respect to its security. Our paper is an attempt at improving the situation. We

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

1. specify a concrete and practical 2AMEX protocol, called 2AMEX-1, reflecting what has been discussed in the various standardization documents, see [NKMHB06,NGG$^+$07],
2. adapt and extend the Bellare-Rogaway framework for analyzing cryptographic protocols to the 2AMEX setting (adjust it to two rounds, incorporate timestamps and payloads with signed parts), and
3. prove the specified protocol secure.

Our work therefore provides a firm theoretical underpinning for realistic *and* secure implementations of services which require authentication.

A simple 2AMEX protocol 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 message obtained, 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. A natural and widely considered reasonable approach to solve this problem is to augment messages by timestamps and use them to filter out replays [CDL06]. 2AMEX-1 follows this approach.

The requirement of authenticated message exchange shows up in very different contexts, while the mechanisms used for implementing it are more or less the same and independent of the respective context. This is reflected in 2AMEX-1: request and reply can both carry arbitrary payloads. We even allow that a payload contains parts signed with keys which are used for securing entire requests and responses, following what web service standards allow [NKMHB06]. Clearly, the use of the keys is restricted for signing parts, because otherwise security is compromised.

Our security analysis is based on the seminal work by Bellare and Rogaway [BR93]. The model developed in this work is, however, not general enough. We extend it in two directions: first, we add timestamps, and, second, we add payloads and signature oracles for dealing with signed parts. A crucial point is that the Bellare-Rogaway framework only considers authentication protocols with at least three rounds and that this is in fact a fundamental requirement for their definition of authenticity; our definition is a non-trivial adaptation of theirs. Another difference is that we carry out a concrete security analysis instead of an asymptotic one; we obtain the latter as a consequence.

Related work. The Bellare-Rogaway framework is only one option for studying computational security of cryptographic protocols. Another widely recognized option are simulation-based frameworks 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 frameworks 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 the aforementioned frameworks would simplify a security proof. A more detailed discussion can be found in our recent work [KSW09], where we treat 2AMEX-1 in a simulation-based framework.

There is a wide range of papers on entity authentication protocols (often in connection with key exchange). Bellare and Rogaway's paper [BR93] 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. In [CCK$^+$08], long-lived principals are dealt with from a complexity point of view, whereas in our work long-lived servers are a modeling issue.

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. As pointed out above, the payloads in 2AMEX protocols are determined by the adversary; in [RS09] a framework will be proposed that models adversarial input in a general fashion.

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 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 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 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 user

Here, r is a randomly chosen message identity which is also used as a handle by the server (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, s rejects if $(t', r, c') \in L$ for some t' and c' or if $t \notin \left[t_{\min} + 1, t_s + \mathrm{tol}_s^+\right]$, and otherwise proceeds as follows: If L contains less than cap_s elements, it inserts (t, r, c) into L. If L contains at least cap_s, 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. The formal description of the protocol can be found in Section 4.

3 Protocol Model

The framework we are working in is an extension and 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 $l_{\text{time}} \in \mathbb{N}$. We also assume there is an arbitrary fixed *identifier set* $\mathrm{IDs} \subseteq \{0,1\}^{l_{\text{ID}}}$ for an arbitrary fixed $l_{\text{ID}} \in \mathbb{N}$ whose elements are called *identifiers*. We use them to identify principals, which can act both as clients and as servers.

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 algorithm which 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 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 $c \in$ IDs and a server $s \in$ 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):

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 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 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.

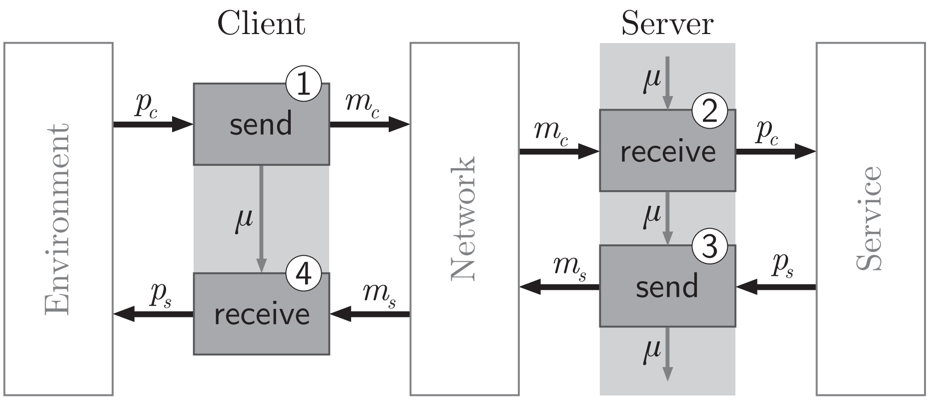

Fig. 1. Message flow in four steps.

Client and Server Algorithms: Input and Output. First, 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, $s \in \text{IDs}$. Second, 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. Third, it gets the local time $t \in \{0,1\}^{l_{\text{time}}}$. Fourth, 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 local 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 local state 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 local state, which it will be provided with the next time it will be called, unless it is reset.

input	client Γ	server Σ
instruction	$\alpha \in \{\mathsf{send}, \mathsf{receive}\}$	$\alpha \in \{\mathsf{send}, \mathsf{receive}, \mathsf{reset}\}$
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		$h \in \{0,1\}^*$
local state	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		$h \in \{0,1\}^*$
local state	μ'	μ'

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

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 receive request, and (iii) accept a 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 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 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 time-stamps and $\{b_j\}_{j\in\mathbb{N}}$ a sequence of bit strings. Assume that for all $i \in \mathbb{N}$ we have

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

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

$\alpha_j \in \{\mathsf{send}, \mathsf{receive}, \mathsf{reset}\}$ and $\mu'_{-1} = \varepsilon$. If for all $i \in \mathbb{N}$ we have

$$\Sigma(\alpha_i, s, pk_{\mathrm{IDs}}, sk_s, t_i, b_i, h_i, \mu_i) = (b'_i, \delta_i, c_i, h'_i, \mu'_i) \tag{2}$$

$$\text{with } \mu_i = \begin{cases} \varepsilon & \text{if } \alpha_i = \mathsf{reset} \\ \mu'_{i-1} & \text{otherwise,} \end{cases} \tag{3}$$

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

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 $\Pi = (\Gamma, \Sigma, \tau, \varphi, E^*)$ where Γ and Σ are the client and server algorithms, τ and φ are the *time* and *freshness functions* (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 supposedly created. The time function will 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 $\varphi(s, \mu_s, t_s)$, 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, in which the protocol is running with an adversary. The latter is simply an arbitrary probabilistic algorithm.

We assume that as a first step the adversary specifies a set of identities $A \subseteq \mathrm{IDs}$, which has to include both the identities of oracles the adversary calls and the identities that will occur in messages. For every principal $s \in \mathrm{IDs}$ a *server instance* Σ_s runs under the identity s. For every pair of principals $c, s \in \mathrm{IDs}$ 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. To set the local clock of a principal, the adversary can use a time instruction with the only restriction that the value of the local clock cannot be decreased by the adversary, i. e., each principal's clock is *monotone*.

There is a *signature oracle* $\mathcal{S}$, which can be used by the adversary (i) to sign bit strings while he constructs the payload for a send instruction, and (ii) to

1. *Generate keys.*
   Let the adversary specify a set $A \subseteq \mathrm{IDs}$, and for each $a \in A$:
   (a) Let $(pk_a, sk_a) \xleftarrow{R} G()$.
   (b) Send (a, pk_a) to the adversary.
2. *Initialize clocks.*
   For each $a \in \mathrm{IDs}$, let $t_a \longleftarrow 0$.
3. *Initialize states and traces 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 $\mathrm{tr}_s \longleftarrow \varepsilon$ and $\mu_s \longleftarrow \varepsilon$.
5. *Initialize step counter.*
   Let $n \longleftarrow 0$.
6. *Run the adversary step by step.*
   Run the adversary, and in each step first increase the counter n and then call client, server or signature algorithm as follows according to the adversary's selection:
   - $\mathit{\Gamma}^i_{c,s}\colon \mathsf{send}(p)$
     (i) $(m, \delta, \mu) \xleftarrow{R} \mathit{\Gamma}(\mathsf{send}, c, s, pk_{\mathrm{IDs}}, sk_c, t_c, p, \mu^i_{c,s})$,
     (ii) $\mu^i_{c,s} \longleftarrow \mu$,
     (iii) $\mathrm{tr}^i_{c,s} \longleftarrow \mathrm{tr}^i_{c,s} \cdot (n, \mathsf{send}, t_c, p, m, \delta)$,
     (iv) return (m, δ, μ) to the adversary.
   - $\mathit{\Sigma}_s\colon \mathsf{receive}(m)$
     (i) $(p, \delta, c, h, \mu) \xleftarrow{R} \mathit{\Sigma}(\mathsf{receive}, s, pk_{\mathrm{IDs}}, sk_s, t_s, m, \varepsilon, \mu_s)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{receive}, t_s, p, m, \delta, c, h)$,
     (iv) return (p, δ, c, h, μ) to the adversary.
   - $\mathit{\Sigma}_s\colon \mathsf{send}(p, h)$
     (i) $(m, \delta, c, h', \mu) \xleftarrow{R} \mathit{\Sigma}(\mathsf{send}, s, pk_{\mathrm{IDs}}, sk_s, t_s, p, h, \mu_s)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{send}, t_s, p, m, \delta, c, h)$,
     (iv) return (m, δ, c, h', μ) to the adversary.
   - $\mathit{\Gamma}^i_{c,s}\colon \mathsf{receive}(m)$
     (i) $(p, \delta, \mu) \xleftarrow{R} \mathit{\Gamma}(\mathsf{receive}, c, s, pk_{\mathrm{IDs}}, sk_c, t_c, m, \mu^i_{c,s})$,
     (ii) $\mu^i_{c,s} \longleftarrow \mu$,
     (iii) $\mathrm{tr}^i_{c,s} \longleftarrow \mathrm{tr}^i_{c,s} \cdot (n, \mathsf{receive}, t_c, p, m, \delta)$,
     (iv) return (p, δ, μ) to the adversary.
   - $\mathit{\Sigma}_s\colon \mathsf{reset}()$
     (i) $(m, \delta, c, h, \mu) \xleftarrow{R} \mathit{\Sigma}(\mathsf{reset}, s, pk_{\mathrm{IDs}}, sk_s, t_s, \varepsilon, \varepsilon, \varepsilon)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{reset}, t_s, \varepsilon, \varepsilon, \mathsf{A}, \varepsilon, \varepsilon)$,
     (iv) return (m, δ, c, h, μ) to the adversary.
   - $\mathcal{S}\colon \mathsf{corrupt}\,(a)$
     (i) $\mathrm{tr}_s \longleftarrow \mathrm{tr}_s \cdot (n, \mathsf{corrupt}, t_s, \varepsilon, \varepsilon, \mathsf{A}, \varepsilon, \varepsilon)$,
     (ii) return sk_a to the adversary.
   - $\mathcal{S}\colon \mathsf{sign}(a, p)$
     (i) If $p \notin E^*$, return $\{p\}_{sk_a}$, otherwise return ε to the adversary.
   - $\mathsf{time}(a, t)$
     (i) $t_a \longleftarrow \max(t_a, t)$,
     (ii) return t_a to the adversary.

Table 2. The experiment $\mathsf{Exp}_{\mathit{\Pi},\mathcal{A}}$ for an adversary $\mathcal{A}$ against a protocol $\mathit{\Pi} = (\mathit{\Gamma}, \mathit{\Sigma}, \tau, \varphi, E^*)$.

corrupt a principal's key. The corresponding instructions are sign and $\mathsf{corrupt}$. 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 perform an action (send, $\mathsf{receive}$, reset, sign, $\mathsf{corrupt}$, time), for which he provides the parameters under his control and receives the output values. The details of this process are given in Table 2.

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 $\mathsf{Exp}_{\Pi,\mathcal{A}}$ for an adversary $\mathcal{A}$ against a protocol Π as above 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 the server algorithm as well as the client algorithm, the time function as well as the freshness function, and the exceptions set. We also fix an arbitrary signature scheme and chose some $l_{\text{nonce}} \in \mathbb{N}$ as the length of the message id's used in the protocol.

Server Algorithm, Freshness Function, and Time Function. Let s be the identity that the server algorithm Σ is called with. As local state 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 $\varphi(s, (t_{\min}, L), t) = \left[t_{\min} + 1, t + \mathrm{tol}_s^+\right]$.

The server first checks if it is called with local state 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.

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 local state $\mu = (t_{\min}, L)$, 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) $t \notin \varphi(s, \mu, t_s)$,
   (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 local state μ is ε, the algorithm randomly chooses the message id $r \xleftarrow{R} \{0,1\}^{l_{\text{nonce}}}$, 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 steps corresponds to steps 2. c)–e) 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 local state is μ, it proceeds as follows:

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

Bit String Representations and Exceptions. Our description above leaves open the actual format of the messages. We assume that the tags (From, …) 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 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 $t \in \{0,1\}^{l_{\text{time}}}$, $r \in \{0,1\}^{l_{\text{nonce}}}$ and $c \in \{0,1\}^{l_{\text{ID}}}$.

4.2 Comments and Caveats

For a fixed protocol run, we will use $t_a(n)$ to denote the value of the local clock of principal a at step n, and $\mu_a(n)$ to denote the local state of the server instance of a before step n.

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 $\varphi(s, \mu_s(n_1), t_s(n_1))$ and $\varphi(s, \mu_s(n_2), t_s(n_2))$ must be disjoint. Due to asynchronous clocks, we need the interval $\varphi(s, \mu_s(n), t_s(n))$ 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 φ 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 l_{nonce}, 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 φ. 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 $t_c(n_1) = \tau(m) \notin \varphi(s, \mu_s(n_2), t_s(n_2))$. 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 local state 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 φ should always be the present time, so that the most successful strategy for the clients is to use truthful timestamps. In Appendix A, 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 use $\mu_s(n)$ to denote the content of the local state μ_s *before* the nth step. We say that for a principal $a \in \mathrm{IDs}$ 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 $\mathrm{tr}^i_{c,s}$ 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 local state information, but no message is altered, no false messages are introduced, and no replay attacks are attempted.

Definition 2. *A 2AMEX protocol Π is (n, ϵ)-*correct *for any benign adversary $\mathcal{A}$ that starts at most n many client sessions, and any $c, s \in \mathrm{IDs}$:*

1. *If $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A}) \in \mathrm{tr}^i_{c,s}$, $(n_2, \mathsf{receive}, t_2, p'_c, m_c, \delta_s, c', h) \in \mathrm{tr}_s$, and $\tau(m_c) \in \varphi(s, \mu_s(n_2), t_2)$, then $c' = c$, $p_c = p'_c$, and $\delta_s = \mathsf{A}$, with probability at least $1 - \epsilon$.*
2. *If 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 $n_2 < n' < n_3$, 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 φ is part of the specification, and a protocol only has to be correct with regard to this choice of φ. Hence a protocol in which φ 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 high probability is that we allow randomness in our algorithms, and therefore collisions cannot be ruled out completely.

5.2 Running Time

For the security definition, we need the notion of running time of algorithms. We will use a probabilistic RAM model based on [CR73], in which arbitrary registers can be accessed in constant time. We also adopt the convention that "time" refers to the actual running time plus the size of the code (relative to some fixed programming language), see, e. g., [BDJR97]. Oracle queries are answered in unit time. We assume that the running time of the algorithms of the signature scheme is as follow: Generating a key pair takes time t_G, signing or verifying a bit-string with l bits takes time $t_S(l)$ or $t_V(l)$, respectively.

5.3 Security Definition

We 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 $\mathsf{NoMatching}_{\Pi,\mathcal{A}}$, which is intended to model the event that the adversary $\mathcal{A}$ has successfully "broken" the protocol Π. 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 $\mathsf{NoMatching}^f_{\Pi,\mathcal{A}}$ (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 $(n_2, \mathsf{receive}, t_2, p_c, m_c, \mathsf{A}, c, h) \in \mathrm{tr}_s$ for which no session number i exists such that the step matches the client trace $\mathrm{tr}^i_{c,s}$ w. r. t. f.

The event $\mathsf{NoMatching}_{\Pi,\mathcal{A}}$ denotes the event that $\mathsf{NoMatching}^f_{\Pi,\mathcal{A}}$ 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, and adversary $\mathcal{A}$, 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 $\mathsf{Adv}_{\Pi,\mathcal{A}} = \Pr\left[\mathsf{NoMatching}_{\Pi,\mathcal{A}}\right]$.

An adversary is called $(t, n_{\mathsf{rcv}}, n_{\mathsf{send}}, n_{\mathsf{sign}}, n_{\mathsf{time}}, n_{\mathsf{cor}}, l_{\text{data}})$*-adversary* if the following holds: Its overall running time is bounded by t; the adversary selects no more than n_{ID} identities in its first step, and for each of these identities, the number of calls with receive, send, sign, or time instructions is bounded by n_{rcv}, n_{send}, n_{sign}, and n_{time}, respectively; in each of these calls, the size of the payload or message provided to the principal is no more than l_{data}; and the total number of principals corrupted by the adversary is not larger than n_{cor}.

Definition 3. *A 2AMEX protocol Π is $(t, n_{\text{ID}}, n_{\mathsf{rcv}}, n_{\mathsf{send}}, n_{\mathsf{sign}}, n_{\mathsf{time}}, n_{\mathsf{cor}}, l_{\text{data}}, \varepsilon)$-*secure *if we have $\mathsf{Adv}_{\Pi,\mathcal{A}} \le \varepsilon$ for any $(t, n_{\text{ID}}, n_{\mathsf{rcv}}, n_{\mathsf{send}}, n_{\mathsf{sign}}, n_{\mathsf{time}}, n_{\mathsf{cor}}, l_{\text{data}})$-adversary $\mathcal{A}$.*

Note that our notion of security also takes care of 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 $(n_1, \mathsf{receive}, t_1, p_c^1, m_c, \mathsf{A}, c, h^1)$ and $(n_2, \mathsf{receive}, t_2, p_c^2, m_c, \mathsf{A}, c, h^2)$, where $n_1 \ne n_2$. 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 $n_1 \ne n_2$, it follows that $i_1 \ne i_2$. 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).—Observe that it is of course allowed for the server to accept the same *payload* twice from the same client.

6 Correctness and Security of 2AMEX-1

First, we state that 2AMEX-1 is indeed correct:

Theorem 4. *The protocol 2AMEX-1 with message id length l_{nonce} is an (n, ϵ)-correct 2AMEX protocol, where $\epsilon = 1 - 2^{-n \cdot l_{\text{nonce}}} \cdot \prod_{i=0}^{n+1}(2^{l_{\text{nonce}}} - i)$.*

Now, following a standard approach, we show that 2AMEX-1 is "secure": For each adversary against 2AMEX-1 we construct an adversary against the underlying signature scheme with comparable running time and success probability. Hence we first recall a standard notion of security for signature schemes: An adversary against a signature scheme is a probabilistic algorithm that as input receives 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 adversary is successful if it produces a pair (m, σ) with a valid signature σ of m (corresponding to pk), and m has not been used as a query for the oracle $\mathcal{O}$. For a running time t, natural numbers q and l, and a probability ϵ, the adversary (t, q, l, ϵ)-breaks the signature scheme if it runs in time bounded by t, uses at most q oracle queries, each query is of length at most l, and is successful with probability at least ϵ. Consequently, a signature scheme is (t, q, l, ϵ)-secure if there is no adversary that (t, q, l, ϵ)-breaks it.

Theorem 5. *2AMEX-1 is* $(t_1, n_{\mathrm{ID}}, n_{\mathsf{rcv}}, n_{\mathsf{send}}, n_{\mathsf{sign}}, n_{\mathsf{time}}, n_{\mathsf{cor}}, l_{\mathrm{data}}, \varepsilon_1)$-secure *if the signature scheme used is* $(t_2, q_2, l_2, \varepsilon_2)$*-secure with*

$$t_2 \in O(t_1 + n_{\mathrm{ID}} \cdot (t_G + n_{\mathrm{ops}} \cdot (\mathrm{cap}_{\max} \cdot (l_{\mathrm{ID}} + l_{\mathrm{time}}) + t_S(l_{\mathrm{msg}})))), \tag{4}$$

$$q_2 \leq n_{\mathsf{sign}} + n_{\mathsf{send}}, \tag{5}$$

$$l_2 \leq l_{\mathrm{msg}}, \tag{6}$$

$$\varepsilon_2 \geq \varepsilon_1 \cdot \frac{1}{n_{\mathrm{ID}}} \cdot \frac{2^{l_{\mathrm{nonce}}}!}{(2^{l_{\mathrm{nonce}}} - n_{\mathrm{ID}} \cdot n_{\mathsf{send}})! \cdot 2^{l_{\mathrm{nonce}} \cdot n_{\mathrm{ID}} \cdot n_{\mathsf{send}}}} \tag{7}$$

where $n_{\mathrm{ops}} = n_{\mathsf{rcv}} + n_{\mathsf{send}} + n_{\mathsf{sign}} + n_{\mathsf{time}}$*, the maximum of the capacities of all servers is* $\mathrm{cap}_{\max}$*, and* $l_{\mathrm{msg}} \in O(l_{\mathrm{ID}} + l_{\mathrm{nonce}} + l_{\mathrm{time}} + l_{\mathrm{data}})$*.*

The security proof for our protocol, see Section 8, first establishes that in 2AMEX-1, no server accepts the same message twice, therefore replay-attacks in their most obvious form are impossible. We then prove that every "break" of our protocol (i.e., every occurrence of $\mathsf{NoMatching}_{\Pi,\mathcal{A}}$) implies that collision of message id's or existential forgery of a signature happened. We then use this fact to construct a simulator that uses an adversary against 2AMEX-1 and a "simulated" protocol environment to construct an adversary against the signature scheme. Theorem 5 then follows from a precise analysis of the resources used and success probability achieved by the thus-obtained adversary. We mention in passing that the constants hidden in the O-notation in Theorem 5 are reasonably small (≤ 100).

Often, a Turing-machine based asymptotic notion of security is considered, where it is required that the success probability of every polynomial-time adversary drops rapidly when the security parameter (in our case, this reflects the length of the keys for the signature scheme as well as the nonce length l_{nonce}) increases. Since Cook and Reckhow proved that RAM machines and Turing machines are polynomially equivalent [CR73], the above Theorem 5 implies the following (see [KSW08] for details):

Corollary 6. *2AMEX-1 is asymptotically secure if the it is used with a signature scheme that is asymptotically secure against existential forgery.*

7 Correctness of 2AMEX-1 (Proof of Theorem 4)

We now prove Theorem 4.

Note that there are $2^{l_{\text{nonce}}}$ different message id's, hence the probability of all of these message id's being different is exactly

$$\frac{2^{l_{\text{nonce}}} \cdot (2^{l_{\text{nonce}}} - 1) \cdot (2^{l_{\text{nonce}}} - 2) \cdot \dots \cdot (2^{l_{\text{nonce}}} - n + 1)}{(2^{l_{\text{nonce}}})^n} , \tag{8}$$

thus ϵ from the statement of the theorem is the probability of a collision of message id's. It therefore suffices to show that the relevant messages are always accepted, unless there are two different client sessions that choose the same message id.

So assume that there are no collisions of message id's, let $(n_1, \mathsf{send}, t_1, p_c, m_c, \mathsf{A}) \in \mathrm{tr}^i_{c,s}$ 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 $t_1 \in \varphi(s, \mu_s(n_2), t_2)$.

First, note that the message id of m_c can only be the same as that of a message that was previously delivered to s if a collision in the above sense occurs, since $\mathcal{A}$ is benign and therefore delivers m_c at most once to s. Hence, we can assume $n_1 < n_2$. We show that none of the four cases that lead to rejection of the message on the server side happens, unless a collision of message id's has occurs. 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 unless a collision appeared, m_c's message id does not already appear in the set L maintained by s. Finally, the message cannot be rejected in step $1(c)$, since by the prerequisites, $\tau(m_c) = t_1 \in \varphi(s, \mu_s(n_s), t_2)$. Thus, the server accepts in all cases where no collision has occurred. 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 $(n', \dots, \mathsf{A}, \dots) \in \mathrm{tr}_s$ 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. □

8 Security of 2AMEX-1 (Proof of Theorem 5)

We now perform a concrete security analysis of 2AMEX-1: We show that an adversary with a given resource bound and success probability against 2AMEX-1 immediately leads to an attack on the signature scheme with resource bound and success probability "close" to the ones of the given adversary against 2AMEX-1.

We proceed in two steps: We first show that every successful attack against 2AMEX-1 must involve the forgery of a signature of an uncorrupted principal, or the collision of two nonces chosen by the client algorithm. Since both of these events happen with very low probability only (provided that the signature scheme is secure), this implies that 2AMEX-1 is secure in an asymptotic sense.

In a second step, for a more detailed analysis, we provide a simulator $\mathcal{S}$ (see Appendix B) which turns any adversary $\mathcal{A}$ against 2AMEX-1 into an adversary $\mathcal{S}_{\mathcal{A}}$ against the signature scheme. We then analyze the success probability of $\mathcal{S}_{\mathcal{A}}$, which is "close" to the success probability of $\mathcal{A}$, and the running time of $\mathcal{S}_{\mathcal{A}}$, which is, roughly speaking, linear in the running time of $\mathcal{A}$.

Note that the first part of the proof does not rely on any assumptions about the security of the signature scheme.

8.1 Attack implies collision or forgery

Theorem 7. *Let $\mathcal{A}$ be an arbitrary adversary. Then for every run of the experiment $\mathsf{Exp}_{2AMEX-1,\mathcal{A}}$ in which the event $\mathsf{NoMatching}_{2AMEX-1,\mathcal{A}}$ occurs, one of the following events occurs as well:*

- *There are two client instances $\Gamma^{i}_{c,s}$ and $\Gamma^{i'}_{c,s}$ with $i \neq i'$, and both client sessions chose the same message id,*
- *$\mathcal{A}$ produced a bitstring that is accepted as a valid signature for an uncorrupted identity a, which was not obtained from the client or server algorithms or the signature oracle.*

Note that to achieve the properties mentioned in the theorem, the client algorithm could also use a counter to determine fresh message id's for each message. This would be sufficient to ensure security of our protocol, but comes with the price of the client having to maintain a long-term state. To prove Theorem 7, 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 8. *Let $\mathcal{A}$ be an adversary and $s \in$ IDs. Then in every run of $\mathsf{Exp}_{2AMEX\text{-}1,\mathcal{A}}$, 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 $n_1 = n_2$.*

For the proof, 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.

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, $t^{s}_{\min}(n_2) < t$ (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. □

Note that the preceding proof is the only situation where we actually use monotonicity of the clocks—it is obvious 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 prove Theorem 7: Fix a run of the experiment $\mathsf{Exp}_{\text{2AMEX-1},\mathcal{A}}$ in which the event $\mathsf{NoMatching}_{\text{2AMEX-1},\mathcal{A}}$ appears. Note that by construction of the experiment, 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: By definition, such messages are elements of the exception set E^*, and hence the signature oracle $\mathcal{S}$ refuses to sign these bitstrings. We now define a partner function as follows: For every client instance $\Gamma^i_{c,s}$, 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 Σ_s with incoming message m, if such a step exists. Let $f(c, s, i)$ be undefined otherwise. In particular, $\mathsf{NoMatching}^f$ appears. By the prerequisites, we know that $\mathsf{NoMatching}$ occurs in the protocol run. Now indirectly assume that neither existential forgery against an uncorrupted key, nor collision of message id's for client sessions $\Gamma^i_{c,s}$ and $\Gamma^{i'}_{c,s}$ for $i \neq i'$ occurs. We distinguish the two cases in the definition of $\mathsf{NoMatching}^f$ (see Section 5.3).

First Case. Assume that case (a) occurs. 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathsf{receive}$-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 $\mathsf{receive}$ 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 between $\Gamma^i_{c,s}$ and $\Gamma^{i'}_{c,s}$ for $i \neq i'$ 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 8, 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 8, 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. □

8.2 Security Proof

We now prove Theorem 5. As noted above, we provide a simulator $\mathcal{S}$ (see Appendix B) which turns an adversary $\mathcal{A}$ against 2AMEX-1 into an adversary $\mathcal{S}_{\mathcal{A}}$ against the signature scheme. By abuse of terminology, we also refer to the adversary $\mathcal{S}_{\mathcal{A}}$ as "the simulator" to distinguish it from the adversary $\mathcal{A}$.

Let $\mathcal{A}$ be a $(t, n_{\mathrm{ID}}, n_{\mathsf{rcv}}, n_{\mathsf{send}}, n_{\mathsf{sign}}, n_{\mathsf{time}}, n_{\mathsf{cor}}, l_{\mathrm{data}})$-adversary against the protocol 2AMEX-1 which has an advantage $\mathsf{Adv}_{\Pi,\mathcal{A}}$. Let $\mathbf{S} = (G, S, V)$ be the signature scheme used in the protocol. We will analyze the adversary $\mathcal{S}_{\mathcal{A}}$ against the signature scheme $\mathbf{S}$. Thus, $\mathcal{S}_{\mathcal{A}}$ will be given a public key $pk_\star$ and a signature oracle $\mathcal{O}_\star$; and to successfully break the signature scheme, it has to provide a message m and a signature σ such that $V((m, \sigma), pk_\star)$ returns true.

We briefly sketch what the simulator does. Roughly speaking, the simulator runs the experiment from Table 2, where it replaces one of the public keys with $pk_\star$. If a message has to be signed with the corresponding private key or if the adversary uses the corresponding signature oracle, the simulator uses $\mathcal{O}_\star$ and logs the signature. If the adversary is successful because it manages to forge a signature which has not been produced by $\mathcal{O}_\star$ and thus not logged, the simulator outputs this forgery. All other queries of the adversary are answered according to the protocol specification.

Success probability First, we analyze the advantage $\mathsf{Adv}_{\mathbf{S},\mathcal{S}_{\mathcal{A}}}$ of the simulator $\mathcal{S}_{\mathcal{A}}$ against the signature scheme $\mathbf{S}$.

Due to Theorem 7 we know that the simulator is successful in breaking the signature scheme if (i) the adversary $\mathcal{A}$ is successful against the protocol Π, (ii) no collision of message id's occurred, (iii) the adversary did not try to corrupt $pk_\star$, and (iv) the adversary forged a signature for the principal that corresponds to key $pk_\star$. Thus, the resulting advantage is the product of the probabilities of these four events.

The probability of a collision of message id's $\mathrm{Pr}_{\mathrm{coll.}}$ can be calculated as follows: For each send instruction at a client, one message id of length l_{nonce} is randomly chosen. Thus, at most $n_{\mathrm{ID}} \cdot n_{\mathsf{send}}$ message id's are chosen from a set of size $2^{l_{\mathrm{nonce}}}$. The resulting probability of a collision is

$$\mathrm{Pr}_{\mathrm{coll.}} = 1 - \frac{2^{l_{\mathrm{nonce}}}!}{(2^{l_{\mathrm{nonce}}} - n_{\mathrm{ID}} \cdot n_{\mathsf{send}})! \cdot 2^{l_{\mathrm{nonce}} \cdot n_{\mathrm{ID}} \cdot n_{\mathsf{send}}}} \quad . \tag{9}$$

The adversary we simulate has access to n_{ID} identities. Each of those has a key pair, and the simulator randomly chooses which of those corresponds to $pk_\star$. Thus, if the adversary corrupts n_{cor} different identities, the probability $\mathrm{Pr}_{\mathrm{corr.}}$ that it does not "hit" the key $pk_\star$ is

$$\mathrm{Pr}_{\mathrm{corr.}} = \prod_{i=0}^{n_{\mathsf{cor}}-1} \frac{n_{\mathrm{ID}} - 1 - i}{n_{\mathrm{ID}} - i} = \frac{n_{\mathrm{ID}} - n_{\mathsf{cor}}}{n_{\mathrm{ID}}} \quad . \tag{10}$$

If the adversary forges a signature, the probability that the corresponding key is $pk_\star$ is

$$\mathrm{Pr}_{\mathrm{key}} = \frac{1}{n_{\mathrm{ID}} - n_{\mathsf{cor}}} \quad . \tag{11}$$

Thus, the overall advantage of the simulator is

$$\begin{aligned} \mathsf{Adv}_{\mathbf{S},\mathcal{S}_\mathcal{A}} &\geq \mathsf{Adv}_{\Pi,\mathcal{A}} \cdot (1 - \mathrm{Pr}_{\mathrm{coll.}}) \cdot \mathrm{Pr}_{\mathrm{corr.}} \cdot \mathrm{Pr}_{\mathrm{key}} & (12) \\ &= \mathsf{Adv}_{\Pi,\mathcal{A}} \cdot \frac{2^{l_{\mathrm{nonce}}}!}{(2^{l_{\mathrm{nonce}}} - n_{\mathrm{ID}} \cdot n_{\mathsf{send}})! \cdot 2^{l_{\mathrm{nonce}} \cdot n_{\mathrm{ID}} \cdot n_{\mathsf{send}}}} \cdot \frac{1}{n_{\mathrm{ID}}} \quad . & (13) \end{aligned}$$

Running Time We now analyze the running time of the simulator $\mathcal{S}_\mathcal{A}$. We first give an asymptotic analysis and then simplify the resulting term for the running time given certain assumptions. First, let $\mathrm{cap}_{\max} = \max\{\mathrm{cap}_s \mid s \in \mathrm{IDs}\}$.

As we use the algorithms of the signature scheme, we use the following variables and functions to denote their running time: Generating a key pair takes t_G time, signing or verifying a bit-string with l bits takes $t_S(l)$ or $t_V(l)$ time, respectively. We assume that $t_V(l) \in O(t_S(l))$ and $t_S(l) \in \Omega(l)$.

We also use maps to store keys and associated values. We assume the time to initialize a new map is constant, we denote the time of the other operations on the map (add, remove, lookup) with $t_{\mathsf{map}}(n, l)$ where n is the maximal number of entries in the map and l is the maximal length of the keys. On the machine model we use, the operations (add, remove, lookup) can be performed in time linear in l, e. g., by using Tries.

Another prerequisite we use is a pair of an encoding function and a decoding function (E, D) which can merge multiple bit strings into a single bit string and extract a number of bit strings from a single bit string, respectively. For each operation mode $(o, n) \in \{(\mathsf{tuple}, 2), (\mathsf{request}, 5), (\mathsf{response}, 4), (\mathsf{signature}, 2)\}$ and all bit strings $\beta_1, \ldots, \beta_n$, we assume $D(\mathsf{o}, E(\mathsf{o}, \beta_1, \ldots, \beta_n)) = (\beta_1, \ldots, \beta_n)$ and $|E(\mathsf{o}, \beta_1, \ldots, \beta_n)| \in O(\sum_{i=1}^n |\beta_i|)$.

Now, the running time of the single functions can be bounded as shown in Tabular 3 for a fixed $l_{\mathrm{msg}} \in O(l_{\mathrm{ID}} + l_{\mathrm{nonce}} + l_{\mathrm{time}} + l_{\mathrm{data}})$. Then the overall

$$
\begin{aligned}
t_{\texttt{time}} &\in O(t_{\texttt{userNr}}) && (14)\\
t_{\texttt{clientSend}} &\in O(t_{\texttt{userNr}} + t_{\texttt{sign}}) && (15)\\
t_{\texttt{serverReceive}} &\in O(t_{\texttt{userNr}} + t_{\texttt{verify}} + \text{cap}_{\max} \cdot (l_{\text{ID}} + l_{\text{time}}) + t_{\texttt{map}}(\text{cap}_{\max}, l_{\text{nonce}})) && (16)\\
t_{\texttt{serverSend}} &\in O(t_{\texttt{userNr}} + t_{\texttt{sign}} + t_{\texttt{map}}(\text{cap}_{\max}, l_{\text{nonce}})) && (17)\\
t_{\texttt{clientReceive}} &\in O(t_{\texttt{userNr}} + t_{\texttt{verify}}) && (18)\\
t_{\texttt{corrupt}} &\in O(t_{\texttt{userNr}}) && (19)\\
t_{\texttt{sign}} &\in O(t_{\texttt{userNr}} + t_S(l_{\text{msg}}) + t_{\texttt{map}}(n_{\mathsf{sign}} + n_{\mathsf{send}}, l_{\text{msg}})) && (20)\\
t_{\texttt{verify}} &\in O(t_{\texttt{userNr}} + t_V(l_{\text{msg}}) + t_{\texttt{map}}(n_{\mathsf{sign}} + n_{\mathsf{send}}, l_{\text{msg}})) && (21)\\
t_{\texttt{userNr}} &\in O(t_{\texttt{map}}(n_{\text{ID}}, l_{\text{ID}})) && (22)
\end{aligned}
$$

Table 3. Running time of the procedures of the simulator

running time of the simulator $\mathcal{S}$ is bounded by the following set, where $n_{\text{ops}} = n_{\mathsf{rcv}} + n_{\mathsf{send}} + n_{\mathsf{sign}} + n_{\mathsf{time}}$:

$$
\begin{aligned}
&O(t_{\mathcal{A}} + n_{\text{ID}} \cdot t_G + n_{\mathsf{cor}} \cdot t_{\texttt{corrupt}} + n_{\text{ID}} \cdot n_{\mathsf{sign}} \cdot t_{\texttt{sign}} + && (23)\\
&\quad n_{\text{ID}} \cdot n_{\mathsf{time}} \cdot t_{\texttt{time}} + n_{\text{ID}} \cdot (n_{\mathsf{send}} \cdot (t_{\texttt{clientSend}} + t_{\texttt{serverSend}}) + \\
&\quad n_{\mathsf{rcv}} \cdot (t_{\texttt{clientReceive}} + t_{\texttt{serverReceive}})) \\
=&O(t_{\mathcal{A}} + n_{\text{ID}} \cdot (t_G + n_{\text{ops}} \cdot (t_S(l_{\text{msg}}) + t_{\texttt{map}}(\text{cap}_{\max}, l_{\text{nonce}}) + && (24)\\
&\quad t_{\texttt{map}}(n_{\mathsf{sign}} + n_{\mathsf{send}}, l_{\text{msg}}) + t_{\texttt{map}}(n_{\text{ID}}, l_{\text{ID}})) + \text{cap}_{\max} \cdot (l_{\text{ID}} + l_{\text{time}}))) \\
=&O(t_{\mathcal{A}} + n_{\text{ID}} \cdot (t_G + n_{\text{ops}} \cdot (t_S(l_{\text{msg}}) + \text{cap}_{\max} \cdot (l_{\text{ID}} + l_{\text{time}})))) && (25)
\end{aligned}
$$

Note that the machine model we use would allow us to address arbitrary registers, e. g., we could directly use bitstrings (encoded as numbers) as register numbers to store or retrieve information and thus replace, e. g., the map which stores information about messages signed so far and their signatures—this would result in an unrealistic speedup for our algorithms and the use of an exponential number of registers in the length of messages. In fact we only addresses in $O(n_{\text{ID}} \cdot n_{\text{ops}} \cdot l_{\text{msg}})$.

Finally, note that the simulator $\mathcal{S}_{\mathcal{A}}$ makes at most $n_{\mathsf{sign}} + n_{\mathsf{send}}$ queries to the signature oracle it is provided with, as this is the maximal number of calls to the `sign` function per identity. In each of these calls, at most l_{msg} are being signed. Thus, the total number of bits signed by the oracle is at most $(n_{\mathsf{sign}} + n_{\mathsf{send}}) \cdot l_{\text{msg}}$. □

9 Conclusion

We provided a model that allows to analyze cryptographic protocols for the practically relevant goal of two-round authenticated message exchange, taking into account common protocol elements such as timestamps, nonces, signatures, and signed parts. For the first time, this allows sound cryptographic security proofs of protocols in this setting. Using our new model we proved secure the protocol 2AMEX-1, which had not been specified in detail before, but variants of which are widely used in practice.

References

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

BDJR97. Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, pages 394–403, 1997.

BEL05. Liana Bozga, Cristian Ene, and Yassine Lakhnech. A symbolic decision procedure for cryptographic protocols with time stamps. *J. Log. Alg. Prog.*, 65(1):1–35, 2005.

BFG04. Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. A semantics for web services authentication. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 198–209. ACM, 2004.

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

BFGP03. Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Riccardo Pucella. Tulafale: A security tool for web services. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 3188 of *LNCS*, pages 197–222. Springer, 2003.

BP03. Michael Backes and Birgit Pfitzmann. A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol. In *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, *CCS 2003*, pages 220–230. ACM, 2003.

BR93. M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. Stinson, editor, *CRYPTO '93*, volume 773 of *LNCS*, pages 232–249. Springer-Verlag, 1993.

Can01. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS 2001*, pages 136–145. IEEE Computer Society, 2001.

CCK$^+$08. Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Nancy A. Lynch, and Olivier Pereira. Modeling computational security in long-lived systems. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *LNCS*, pages 114–130. Springer, 2008.

CDL06. Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *J. Comput. Sec.*, 14(1):1–43, 2006.

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 *LNCS*, pages 380–403. Springer, 2006.

CR73. Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

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

DS81. Dorothy E. Denning and Giovanni M. Sacco. Timestamps in key distribution protocols. *Comm. 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 *LNCS*, 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.

KSW08. Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. Computationally secure two-round authenticated message exchange. Technical Report 0809, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 2008.

KSW09. Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. A simulation-based treatment of authenticated message exchange. In *ASIAN*, 2009. To appear.

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.

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 *LNCS*, pages 147–166. Springer, 1996.

ML07. Nilo Mitra and Yves Lafon. SOAP version 1.2 part 0: Primer (second edition). Technical report, W3C, 2007.

NGG$^+$07. Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. WS-SecurityPolicy 1.2. Technical report, OASIS Web Services Secure Exchange TC, 2007. OASIS Standard.

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. *Comm. ACM*, 21(12):993–999, 1978.

RS09. Phillip Rogaway and Till Stegers. Authentication without elision. In *CSF*. IEEE Computer Society, 2009. To appear.

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. *J. Comput. Sec.*, 13(3):565–591, 2005.

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

A Choice of 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 φ 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 9. *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 $t_{\min}^s \leq t_s - \mathrm{tol}_s^-$.

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.

Assume the last reset (or initialization, which for the server is the same event) of s happened at the time t_r^s (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 $M = (t_i^c, t_i^s)_{i\in\mathbb{N}}$, where a pair (t_i^c, t_i^s) represents a message that the server s receives at time t_i^s, and which carries the client's timestamp t_i^c. Since the minimal delay between incoming messages and between a reset and an incoming message is t_{diff}, we require that $t_r^s + t_{\text{diff}} \leq t_0^s$, and $t_i^s + t_{\text{diff}} \leq t_{i+1}^s$ for all i. We also require that $t_i^c \leq t_i^s + \text{tol}_s^+$ for all i (other sequences cannot be accepted by the server). With $t_{\min}^s(M)(t^s)$ 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 $t_{\min}^s(M)(t^s)$ 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 $t_{\min}^s(M)(t^s)$, 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* $M_c = (t_r^s + i \cdot t_{\text{diff}}, t_r^s + \text{tol}_s^+ + i \cdot t_{\text{diff}})_{i\geq 1}$. 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:

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

We first consider the case $t = t_r^s + \text{tol}_s^+ + \text{tol}_s^-$. In this case, exactly $\text{tol}_s^+ + \text{tol}_s^-$ units of time have passed since the last reset. In this time, s has accepted exactly $(\text{tol}_s^+ + \text{tol}_s^-)/t_{\text{diff}}$ 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 $t_r^s + \text{tol}_s^+$ by the specification of the protocol. Hence $t_{\min}^s(M_c)(t) = t_r^s + \text{tol}_s^+ = t - \text{tol}_s^-$, which proves the required inequality. For points in time beyond $t = t_r^s + \text{tol}_s^+ + \text{tol}_s^-$, 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. $\square$

B The Simulator

We now define the simulator $\mathcal{S}$. The addition on time values, i. e. on l_{time} bit numbers, is denoted by $\dot{+}$. We assume the simulator is provided with a public key $pk_\star$ and a signature oracle $\mathcal{O}_\star$ which it is supposed to attack, it has access to the capacities and tolerances of the servers (i. e., to cap_s and tol_s^+ for each $s \in \text{IDs}$), the signature scheme (G, S, V), and the encoding and decoding functions (E, D).

```
main
let $u = 0$
let $U = \texttt{newMap}()$
let $M = \texttt{newMap}()$
let $\mathcal{A}$ choose a set $A \subseteq \text{IDs}$ with $|A| = n_{\text{ID}}$
choose $x \leq n_{\text{ID}}$ at random
for $a \in A$
    let $\bar{a} = \texttt{userNr}(a)$
    let $t_{\bar{a}} = 0$
    if $\bar{a} = x$,
        let $pk_x = pk_\star$
    else,
        let $(pk_{\bar{a}}, sk_{\bar{a}}) = G()$
    send $(a, pk_{\bar{a}})$ to the adversary
simulate $\mathcal{A}$
    if $\mathcal{A}$ sends $\mathsf{time}(a, t)$
        return $\texttt{time}(a, t)$
    if $\mathcal{A}$ sends $\mathsf{send}(p)$ to $\mathit{\Gamma}_{c,s}^i$
        return $\texttt{clientSend}(c, s, i, p)$
    if $\mathcal{A}$ sends $\mathsf{receive}(m)$ to $\mathit{\Sigma}_s$
        return $\texttt{serverReceive}(s, m)$
    if $\mathcal{A}$ sends $\mathsf{send}(p, h)$ to $\mathit{\Sigma}_s$
        return $\texttt{serverSend}(s, p, h)$
    if $\mathcal{A}$ sends $\mathsf{receive}(m)$ to $\mathit{\Gamma}_{c,s}^i$
        return $\texttt{clientReceive}(c, s, i, m)$
    if $\mathcal{A}$ sends $\mathsf{corrupt}(a)$ to $\mathcal{S}$
        return $\texttt{corrupt}(a)$
    if $\mathcal{A}$ sends $\mathsf{sign}(a, p)$ to $\mathcal{S}$
        if $D(\mathsf{request}, p)$ or $D(\mathsf{response}, p)$ is successful
            return $\varepsilon$
        return $\texttt{sign}(a, p)$

time$(a, t)$
let $\bar{a} = \texttt{userNr}(a)$
if $t \geq t_{\bar{a}}$, set $t_{\bar{a}} = t$
return $t_{\bar{a}}$
```

```
$\underline{\texttt{clientSend}(c, s, i, p)}$
let $\bar{c} = \texttt{userNr}(c)$ and $\bar{s} = \texttt{userNr}(s)$
if $\mu^{i}_{\bar{c},\bar{s}} \neq \varepsilon$
    return $(\varepsilon, 0)$
let $r$ be a random $l_{\text{nonce}}$-bit number
let $\mu^{i}_{\bar{c},\bar{s}} = r$
let $m = E(\textsf{request}, c, s, r, t_{\bar{c}}, p)$
let $\sigma = \texttt{sign}(c, m)$
let $\hat{m} = E(\textsf{signature}, m, \sigma)$
return $(\hat{m}, 1)$

$\underline{\texttt{serverReceive}(s, \hat{m})}$
let $\bar{s} = \texttt{userNr}(s)$
if $\mu^{t_{\min}}_{\bar{s}} = \varepsilon$
    let $\mu^{t_{\min}}_{\bar{s}} = t_{\bar{s}} \dotplus \mathrm{tol}^{+}_{s}$ and $\mu^{L}_{\bar{s}} = \texttt{newMap}()$
try
    let $(m, \sigma) = D(\textsf{signature}, \hat{m})$
    let $(c, s', r, t, p) = D(\textsf{request}, m)$
if any error occurred while decoding
    or $s' \neq s$ or $\texttt{verify}(c, m, \sigma) = 0$
    or $t \leq \mu^{t_{\min}}_{\bar{s}}$ or $t > t_{\bar{s}} \dotplus \mathrm{tol}^{+}_{s}$ or $\texttt{lookup}(\mu^{L}_{\bar{s}}, r) \neq \varepsilon$
    return $(\varepsilon, 0, \varepsilon, \varepsilon)$
if $\texttt{size}(\mu^{L}_{\bar{s}}) \geq \mathrm{cap}_s$
    let $\mu^{t_{\min}}_{\bar{s}} = t_{\bar{s}} \dotplus \mathrm{tol}^{+}_{s}$
    for $v$ in $\texttt{allValues}(\mu^{L}_{\bar{s}})$
        let $(t', a) = D(\textsf{tuple}, v)$
        if $t' < \mu^{t_{\min}}_{\bar{s}}$, let $\mu^{t_{\min}}_{\bar{s}} = t'$
    for $v$ in $\texttt{allValues}(\mu^{L}_{\bar{s}})$
        let $(t', a) = D(\textsf{tuple}, v)$
        if $t' \leq \mu^{t_{\min}}_{\bar{s}}$, $\texttt{remove}(T, v)$
$\texttt{add}(T, r, E(\textsf{tuple}, t, c))$
return $(p, 1, c, r)$

$\underline{\texttt{serverSend}(s, p, h)}$
let $\bar{s} = \texttt{userNr}(s)$
if $\mu^{t_{\min}}_{\bar{s}} = \varepsilon$
    let $\mu^{t_{\min}}_{\bar{s}} = t_{\bar{s}} \dotplus \mathrm{tol}^{+}_{s}$ and $\mu^{L}_{\bar{s}} = \texttt{newMap}()$
let $v = \texttt{lookup}(\mu^{L}_{\bar{s}}, h)$
if $v = \varepsilon$, return $(\varepsilon, 0, \varepsilon, \varepsilon, \mu_{\bar{s}})$
let $(t, c) = D(\textsf{tuple}, v)$
if $c = \varepsilon$, return $(\varepsilon, 0, \varepsilon, \varepsilon, \mu_{\bar{s}})$
$\texttt{remove}(\mu^{L}_{\bar{s}}, h)$
$\texttt{add}(\mu^{L}_{\bar{s}}, h, E(\textsf{tuple}, t, \varepsilon))$
let $m = E(\textsf{response}, s, c, r, p)$
let $\sigma = \texttt{sign}(s, m)$
let $\hat{m} = E(\textsf{signature}, m, \sigma)$
return $(\hat{m}, 1, c, \varepsilon)$
```

```
clientReceive$(c, s, i, \hat{m})$
let $\bar{c} = \texttt{userNr}(c)$ and $\bar{s} = \texttt{userNr}(s)$
if $|\mu^i_{\bar{c},\bar{s}}| \neq l_{\text{nonce}}$, return $(\varepsilon, 0, \mu^i_{\bar{c},\bar{s}})$
try
    let $(m, \sigma) = D(\textsf{signature}, \hat{m})$
    let $(s', c', r, p) = D(\textsf{response}, m)$
if any error occurred while decoding or $c' \neq c$
    or $s' \neq s$ or $\texttt{verify}(s, m, \sigma) = 0$
    or $r \neq \mu^i_{\bar{c},\bar{s}}$
    return $(\varepsilon, 0)$
let $\mu^i_{\bar{c},\bar{s}} = 0^{l_{\text{nonce}}+1}$
    return $(p, 1)$

corrupt$(a)$
let $\bar{a} = \texttt{userNr}(a)$
if $\bar{a} = x$
    stop the simulation, but return no forgery
return $sk_{\bar{a}}$

sign$(a, \beta)$
let $\bar{a} = \texttt{userNr}(a)$
if $\bar{a} \neq x$
    return $S(\beta, sk_{\bar{a}})$
else
    let $\sigma = \mathcal{O}_\star(\beta)$
    $\texttt{add}(M, \beta, \sigma)$
    return $\sigma$

verify$(a, \beta, \sigma)$
let $\bar{a} = \texttt{userNr}(a)$
let $b = V(\beta, \sigma, pk_{\bar{a}})$
if $\bar{a} = x$ and $b = 1$ and $\texttt{lookup}(M, \beta) = \varepsilon$
    stop the simulation and return $(\beta, \sigma)$ as a forgery
return $b$

userNr$(a)$
let $\bar{a} = \texttt{lookup}(U, a)$
if $\bar{a} = \varepsilon$
    let $\bar{a} = u$
    let $u = u + 1$
    $\texttt{add}(U, a, \bar{a})$
return $\bar{a}$
```