Secure Two-Round Message Exchange

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Klaas Ole Kürtz

Kiel
2010

Erster Gutachter: Prof. Dr. Thomas Wilke, Christian-Albrechts-Universität zu Kiel
Zweiter Gutachter: Prof. Dr. Ralf Küsters, Universität Trier

Datum der mündlichen Prüfung: 10. Dezember 2010

Abstract

Cryptographic protocols are an integral part of today's communication infrastructure, where information has to be secured on various levels. The security of cryptographic protocols has been studied for decades with numerous methods, focuses, and levels of abstraction. However, no model has been developed so far that captures the specifics of message exchange protocols that only consist of two rounds, a single client request and a subsequent server response.

In this thesis, we define and analyze three protocols that offer security guarantees for two-round message exchange protocols, including authenticated message exchange (using digital signatures or passwords) as well as confidentiality (using hybrid encryption). Our protocols are generic in the sense that they can be used to securely implement any service based on two-round message exchange, because request and response can carry arbitrary payloads. Our modelings and analyses include realistic aspects characteristic to secure two-round protocols like timestamps and long-lived, but bounded server memory.

All three protocols are uniformly modeled and analyzed in a simulation-based security framework, allowing us to modularize the security analyses and allowing others to build upon our protocols while utilizing the strong composability results offered by the simulation-based security notion. In addition, one of the protocols is analyzed in a concrete computational model; for this we extend the Bellare–Rogaway framework by, e. g., timestamps and payloads with signed parts.

Analyzing one of the protocols in both a simulation-based and a concrete computational framework gives insights into the relation between both frameworks; we show a connection for mutual authentication protocols and point out differences for the case of secure two-round message exchange. In addition, we discuss how our goal of authenticated message exchange is related to the common security notions of message and entity authentication.

Contents

1. Introduction **9**

2. Secure Two-Round Message Exchange **21**
2.1. Prerequisites 21
2.1.1. Two-Round Protocols 21
2.1.2. Asymptotic Analyses 22
2.1.3. Cryptographic Primitives 22
2.2. Notions of Authentication 27
2.3. Securing Two-Round Message Exchange 30
2.3.1. Message Wrapping and Alternative Approaches 30
2.3.2. Features of our Models 31
2.4. Protocol Classes 33
2.4.1. Signature-Authenticated Two-Round Message Exchange 34
2.4.2. Confidential Signature-Authenticated Two-Round Message Exch. 34
2.4.3. Password-Authenticated Two-Round Message Exchange 34
2.5. Proposed Protocols 35
2.5.1. Signature-Authenticated Two-Round Message Exchange 35
2.5.2. Confidential Signature-Authenticated Two-Round Message Exch. 36
2.5.3. Password-Authenticated Two-Round Message Exchange 37

3. Trace-Based Analysis **39**
3.1. The Bellare–Rogaway Framework 39
3.2. Protocol Model 41
3.2.1. Prerequisites 41
3.2.2. Clients and Servers 42
3.2.3. Protocols, the Adversary, and the Experiment 45
3.3. The Protocol SA2ME-1 48
3.3.1. Formal Definition of the Protocol 48
3.3.2. Comments and Caveats 50
3.4. Correctness and Security Definitions 52
3.4.1. Correctness Definition 52
3.4.2. Security Definition 53
3.5. Correctness and Security of SA2ME-1 55
3.5.1. Correctness of SA2ME-1 56
3.5.2. Security of SA2ME-1 57
3.6. Practical Choice of Parameters 64

4. Simulation-Based Analysis **67**
4.1. Simulation-Based Security and the IITM Framework 67
4.1.1. Inexhaustible Interactive Turing Machines 67
4.1.2. Systems of IITM's for Cryptographic Protocols 69
4.1.3. Protocol Security in the IITM Framework 70
4.1.4. Notation for IITM's 71
4.2. Ideal Functionality for Secure Two-Round Message Exchange 73
4.2.1. Parameters 74
4.2.2. Analyzing Password-Based Security 75
4.2.3. Regular Operation and the Interface for the Environment 76
4.2.4. Attacks and the Interface for the Adversary 78
4.2.5. Corruption 80
4.3. Realizing Secure Two-Round Message Exchange 81
4.3.1. Prerequisites and Used Functionalities 81
4.3.2. Signature-Authenticated Two-Round Message Exchange 88
4.3.3. Confidential Signature-Authenticated Two-Round Message Exch. 91
4.3.4. Password-Authenticated Two-Round Message Exchange 93
4.4. Results and Proofs 95
4.4.1. Signature-Authenticated Two-Round Message Exchange 96
4.4.2. Confidential Signature-Authenticated Two-Round Message Exch. 101
4.4.3. Password-Authenticated Two-Round Message Exchange 105
4.5. Implementing the Protocols 110
4.5.1. Uniform and Non-Uniform Adversaries 110
4.5.2. Signature Functionality $\mathcal{F}_{\text{SIG}}$ 111
4.5.3. Signature Interface $\mathcal{P}_{\text{SI}}$ 111
4.5.4. Encryption Functionality $\mathcal{F}_{\text{ENC}}$ 112
4.5.5. The Key Store Functionalities $\mathcal{F}_{\text{KS}^{\text{sig}}}$ and $\mathcal{F}_{\text{KS}^{\text{ae}}}$ 113
4.5.6. The Local Clock Functionality $\mathcal{F}_{\text{LC}}$ 114
4.5.7. Implementing the Protocols 114
4.6. Comments and Caveats 115
4.6.1. Roles of the Environment and the Adversary 115
4.6.2. Correctness Definition 116
4.6.3. Technicalities 118
4.6.4. Joint State Realizations 121
4.6.5. Comparison with Pre-Published Results 122

5. Relation between the Two Frameworks **125**
5.1. Mutual Authentication 125
5.1.1. Ideal Functionalities 125
5.1.2. Implementing Mutual Authentication 126
5.1.3. Bellare–Rogaway Security Implies Secure Realization 128
5.1.4. Secure Realizations do not Yield Secure Bellare–Rogaway Prot. . 131
5.1.5. Extension to Authenticated Key Exchange 131

5.2. Secure Two-Round Message Exchange 132
5.2.1. Problems When Relating Both Models 132
5.2.2. Matching Conversations 133

6. Conclusion **135**

A. The Simulator for the Trace-Based Analysis **139**

B. IITM's for Secure Two-Round Message Exchange **143**
B.1. Ideal Functionality 143
B.1.1. Message Exchange Functionality $\mathcal{F}_{\text{MX}}$ 143
B.1.2. Server Management Functionality $\mathcal{F}_{\text{SM}}$ 145
B.1.3. Enriching Input Functionality $\mathcal{F}_{\text{EI}}$ 146
B.2. Realization 146
B.2.1. Client Functionality (SA) $\mathcal{P}_{\text{C}}^{\text{SA}}$ 146
B.2.2. Client Functionality (CSA) $\mathcal{P}_{\text{C}}^{\text{CSA}}$ 147
B.2.3. Client Functionality (PA) $\mathcal{P}_{\text{C}}^{\text{PA}}$ 148
B.2.4. Server Functionality (SA) $\mathcal{P}_{\text{S}}^{\text{SA}}$ 149
B.2.5. Server Functionality (CSA) $\mathcal{P}_{\text{S}}^{\text{CSA}}$ 150
B.2.6. Server Functionality (PA) $\mathcal{P}_{\text{S}}^{\text{PA}}$ 152
B.2.7. Signature Key Store Functionality $\mathcal{F}_{\text{KS}^{\text{sig}}}$ 154
B.2.8. Public Key Encryption Key Store Functionality $\mathcal{F}_{\text{KS}^{\text{ae}}}$ 155
B.2.9. Signature Interface Functionality $\mathcal{P}_{\text{SI}}$ 155
B.2.10. Signature Interface Dummy Realization $\mathcal{P}_{\text{SI}}^{\text{dummy}}$ 156
B.2.11. Local Clock Functionality $\mathcal{F}_{\text{LC}}$ 156
B.2.12. Random Oracle Functionality $\mathcal{F}_{\text{RO}}$ 156
B.3. Simulators 157
B.3.1. Simulator (SA) $\mathcal{S}_{\text{S2ME}}^{\text{SA}}$ 157
B.3.2. Simulator (CSA) $\mathcal{S}_{\text{S2ME}}^{\text{CSA}}$ 160
B.3.3. Simulator (PA) $\mathcal{S}_{\text{S2ME}}^{\text{PA}}$ 165

C. IITM's for Mutual Authentication **171**
C.1. Ideal Functionality 171
C.1.1. Ideal Single-Session Mutual Authentication Functionality $\mathcal{F}_{\text{MA}}^{\text{SS}}$. 171
C.1.2. Ideal Multi-Session Mutual Authentication Functionality $\mathcal{F}_{\text{MA}}^{\text{MS}}$ . . 171
C.2. Realization 172
C.2.1. Single-Session Protocol Wrapper $\mathcal{P}_{\Pi}^{\text{SS}}$ 172
C.2.2. Multi-Session Protocol Wrapper $\mathcal{P}_{\Pi}^{\text{MS}}$ 172
C.2.3. Single-Session Key Generator Wrapper $\mathcal{P}_{\mathcal{G}}^{\text{SS}}$ 173
C.2.4. Multi-Session Key Generator Wrapper $\mathcal{P}_{\mathcal{G}}^{\text{MS}}$ 173
C.3. Simulator $\mathcal{S}_{\text{MA}}^{\text{MS}}$ 174
C.4. Corruption Macro `Corr` 175

Bibliography **177**

1. Introduction

A characteristic feature of certain standardized communication protocols (such as web services or remote procedure calls, see, e. g., [ML07, Sun98, Win99]) is their restricted form of communication, 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. We call those protocols two-round message exchange protocols.

Several security issues arise from this setting. While one generic approach to address such issues is to run it over a standardized security protocol like TLS [DA06] or SSH [Ylö96], it is sometimes more appropriate to secure the messages directly, i. e., enrich them with security-related parts or wrap them in a secure "container" message. In this way, the messages are not only secured during transport, but they can be stored or passed on including their security features. We follow the latter approach.

One of the security goals arising is that of authenticated message exchange: The server wants to be convinced that the request originated from the alleged client and is not an (unauthorized) duplicate, and the client wants to be convinced that the response originated from the server and is a response to its request; our models will later allow us to express this formally. This goal of authenticated message exchange can be achieved using digital signatures; however, it is sometimes useful that at least the client may use a password instead of digital signatures. Therefore, we will later analyze not only protocols in which signatures are used for authentication, but also a protocol in which only servers use signatures, whereas clients use passwords for authentication.

Another security goal is that of confidentiality: Both parties want to be sure that only the two parties involved in a protocol run can read the information they exchange; one of our protocols will achieve this goal using hybrid encryption.

Analyzing the security of such cryptographic protocols is a widely developed research area, see the next section for an overview. However, so far no model has captured the specifics of two-round message exchange protocols. And although two-round protocols for the above-mentioned goals have been alluded to in the literature, to our knowledge, our work is the first to formally and rigourously specify and analyze protocols for these scenarios.

For the analysis, we build upon frameworks developed for analyzing cryptographic protocols: Based on [BR93a], we develop a model for analyzing secure two-round message exchange protocols that offer signature-based authentication; and using [Kü06a], we model and analyze three two-round message exchange protocols that offer different security guarantees. This requires us to 1. model features in both frameworks that are usually not included in protocol models, e. g., timestamps and long-term memory, and

2. fine-tune the security guarantees we can give—for example, the standard definition for authentication from [BR93a] is just applicable for three or more rounds and cannot be trivially adapted to two rounds.

Thus, this thesis provides the first formal treatment of two-round protocols that faithfully includes relevant characteristics of such protocols, we prove secure three concrete protocols, and we gain insights into notions of authentication as well as the frameworks we build upon for our analyses.

State of the Art and Related Work

Although cryptography has been studied for thousands of years, designing secure cryptographic protocols still seems to be stunningly error-prone. The most prominent example is the Needham–Schroeder authentication protocol presented in [NS78], which was years later found to be vulnerable to a seemingly obvious attack, see [Low96]. During the last decades, the formal analysis of the security of cryptographic protocols has been a vast research area.

Analysis of Cryptographic Protocols

Numerous security notions for cryptographic protocols have been defined on various levels of generality and abstraction. For some of these notions, tools for automatic analyses are developed (or, first, (un)decidability results are shown), while other notions are used to prove protocols secure by hand. Some notions are directed at a specific class of cryptographic protocols, while other notions provide a general framework to model multiple security goals.

Symbolic and Computational Analysis Many abstract—also called *symbolic*—models are based on the work of Dolev and Yao [DY83], who treat messages as formal terms and essentially model cryptographic primitives as perfect "black boxes". The advantage of this approach is that automatic analysis of many security properties is possible on this level, see, e. g., [RT03]. However, there are limits to the automatic analysis of cryptographic protocols: For example, most security properties become undecidable when an unbounded number of sessions is considered [DLMS04].

In contrast to symbolic models, *computational* models [BR93a, Kü06a, Can01, BPW03] treat protocol participants (as well as the adversary) as probabilistic algorithms and view messages as bit strings. Automated analysis in such models is, in general, quite complicated, with [Bla07] being a notable exception. There are, however, some results that show how security statements for a symbolic model can be extended to a computational model [AR02, CH06, BL06].

Computational models may be of *asymptotic* or *concrete* nature. Concrete analyses like [BDJR97] precisely compute the success probability of adversaries depending on resource bounds, while asymptotic analyses like [BR93a] make more abstract statements,

e. g., that the success probability of adversaries can be decreased super-polynomially (up to being "negligible") by increasing the bit length of cryptographic keys.

All work in this thesis is done in computational models; the first analysis is a concrete one, the second analysis is of asymptotic nature.

The Bellare–Rogaway Framework The analysis in Section 3 is based on the Bellare–Rogaway framework from [BR93a], which introduced the first provably secure protocol for entity authentication and authenticated key exchange. Their approach to defining a security notion is called *trace-based* as the (in)security of a protocol is defined based on logs (or "traces") of the actions of an adversary interacting with the protocol. The framework has been used, e. g., in [War05] for a computational analysis of the Needham–Schroeder–Lowe entity authentication protocol, or in [Cho07] to prove secure a revised version of the Yahalom protocol.

There are also various papers that extend the work of Bellare and Rogaway in multiple directions. For example, in [BR95], the framework is extended to handle the three-party case and explicitly distinguishes between "normal" players and a server. This model was then extended in [BPR00] to analyze password-based protocols, and this model was in turn extended to the setting of group protocols in [BCPQ01]. The authors of [BWM97] extended the original framework to the asymmetric setting. In [CK01], the framework was combined with the adversarial model of [BCK98] to use key exchange protocols for the construction of secure channels. This model has then be extended in [LLM07] to capture additional attacks.

We note that most extensions concern the security of authenticated key exchange, while our work is based on the modeling of entity authentication (with the exception of Chapter 5). We also mention the work in [BF09], in which—independently of our work and with a different focus—the Bellare–Rogaway framework is extended to model timestamps.

When referencing [BR93a] it is important to note that their work contains a serious flaw addressed by Charles Rackoff in unpublished work and later addressed and corrected in [Sho99]; but again, this concerns the security of authenticated key exchange, not entity authentication.

Simulation-Based Security A subclass of the computational models are the *simulation-based* models such as Canetti's *Universal Composition* framework [Can01], Backes, Pfitzmann, and Waidner's *Black-Box Reactive Simulatability* framework [PW01, BPW07] or Küsters' *Inexhaustible Interactive Turing Machine (IITM)* framework [Kü06a]. These frameworks "guarantee security even when a secure protocol [...] is used as a component of an arbitrary system" [Can01] and they enable "modular proofs of security" [PW01].

More precisely, these frameworks allow us to abstract from the implementational details of cryptographic primitives and use idealized variants called *ideal functionalities*. The general mechanism is to provide such an ideal functionality for a cryptographic

primitive and prove that (with very high probability) each attack against the real cryptographic implementation can be translated into an attack against the ideal functionality. Together with general compositional results this provides a way to modularly compose a secure protocol from secure cryptographic primitives.

The Universal Composition framework [Can01] introduced the idea of providing a unified and standardized way of capturing the security guarantees of a cryptographic task in ideal functionalities and then proving general security-preserving composition results that allow real cryptographic implementations to replace the ideal functionalities step-by-step.

Küsters' IITM framework [Kü06a] follows the same approach and uses interactive Turing machines to model both ideal functionalities as well as real cryptographic implementations. It offers a more precise handling of the running time of machines (thus, the *inexhaustible* in IITM, where a central point in the proof is how machines with polynomial running time can be combined into a single machine with polynomial running time) and a natural joint-state theorem (which allows to analyze multiple sessions independently, but to later join their states in a secure way). A more detailed discussion on similarities and differences to the Universal Composition framework can be found in [Kü06a].

Other simulation-based security frameworks include Backes, Pfitzmann, and Waidner's *Black-Box Reactive Simulatability* [PW01, BPW07] mentioned above (which uses probabilistic polynomial-time IO automata to model protocols and machines), Delaune, Kermer, and Pereira's framework [DKP09] (which is of symbolic nature and based on the applied pi calculus), or Backes, Pfitzmann, and Waidner's *Cryptographic Library* [BPW03] (which offers both an idealized library of cryptographic primitives as well as an implementation of that library that is proven to securely realize the idealized library).

In [KDMR08], the authors prove results comparing and linking the security notions of several above-mentioned simulation-based frameworks. For further notes on (the history of) the simulation-based approach, we refer the reader to [BPW07].

Several ideal functionalities for standard cryptographic primitives have been defined and proven secure in the simulation-based sense: For example, public-key encryption is dealt with in [CKN03, KT08a], symmetric encryption and authenticated symmetric encryption are treated in [KT09a], digital signatures are dealt with in [Can04, BH04, KT08a], and key derivation for encryption is studied in [KT10].

There are, however, only few complex cryptographic protocols that have been analyzed within the simulation-based framework. We are aware of [CK02, MN06, BCJ+06, BP06a, GMP+08, GBN09, RKP09], where, for instance, Kerberos and the Yahalom protocol are treated. Entity authentication has also been studied in the Universal Composition framework [CH06] and in combination with the Cryptographic Library [BP03].

In [Sho99], an interesting connection is shown between the security definition for authenticated key exchange from [BR93a] and a simple simulation-based security definition.

Our analysis in Section 4 uses Küsters' IITM framework [Kü06a].

Web Services and Web Service Security

The initial motivation for our work and the most prominent examples for two-round protocols that we use throughout this thesis are *web services*, and as discussed later on, our models adopt some features and conditions from the web services setting. In addition, the protocols we propose have partially been alluded to in the web service world. Thus, we give an overview of the work in this area with respect to security.

Web services are defined in [HB04] as

> [...] a software system designed to support interoperable machine-to-machine interaction over a network. [...] Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Herein, *SOAP* refers to the message format standard defined in [ML07, NGM+07, KMG+07] for the exchange of messages encoded in the *Extensible Markup Language (XML)* [BPSM+08].

Web services are not itself a new technology, the main goal of web service technologies is to offer a standardized and modularized way to design network protocols.

For example, the XML standard covers the signaling of the character encoding in messages, *XML Schema* is employed for (extendable) data types, the structure of messages is defined in the SOAP standard, the interface of a web service can be described in the *Web Service Description Language (WSDL)* and so on. The standardization does not only allow interoperability, but also to provide software support for developers that want to offer or call web services. The modularization allows, e. g., to separate aspects like security or message transport from other aspects of a network protocol.

Web Service Security Standards Consequently, there exist extensive standards for incorporating security mechanisms such as encryption, digital signatures, key derivation, etc. into web services. First, *XML Signature* [SRE02] and *XML Encryption* [ER02] define mechanisms to digitally sign or encrypt XML documents in a way that accounts for the specifics of XML.[1] Then, *WS-Security* [NKMHB06] defines mechanisms which builds upon XML Signature and XML Encryption to secure SOAP messages; more precisely, WS-Security defines elements that can be inserted in the header of a SOAP message to carry a signature or to indicate signed and encrypted parts etc.

Other related standards exist, e. g., for policies that express which security mechanisms a server expects on incoming messages or incorporates into outgoing messages [NGG+07b], for establishing secure sessions [NGG+07a], or for requesting and issuing security tokens [NGG+07c]. But some of these standards are so flexible that so-called *profiles* like the *Basic Security Profile* [MGMB10] have been defined to ease the development of interoperable implementations by limiting the flexibility.

[1] We note that [BFH+01] defines an alternative, but apparently outdated standard to embed signatures in SOAP messages.

Web Service Security Research Due to the intended flexibility, neither WS-Security nor any other standard or profile defines a specific protocol in the cryptographic sense, e. g., specify what parts of a SOAP message should be signed or encrypted etc., let alone provide any kind of security guarantee. In contrast, the goal of WS-Security and related work is to enable the designer of a protocol to chose the security mechanisms fitting its security goals and surrounding conditions.

Hence, the underlying security issues and mechanisms for dealing with them are discussed in various papers. Since 2004, there is an annual *ACM Workshop on Secure Web Services* [DM04, DM05, DG06, DP07, DP08, DPS09]; thus, we can only provide an insight into the work on this topic.

First, we mention that [KR06] proposed a way to map protocols based on WS-Security to "all the methods [...] that have been developed in the last decade by the theoretical community for the analysis of cryptographic protocols", and they argue that this mapping preserves flaws and attacks. But as [BG05] shows, there are several specifics of web services that are usually not considered in other protocol models, which weakens the statement in [KR06]. The specifics listed in [BG05] include running multiple protocols with the same identity, shared access to cryptographic keys, password-based authentication, or timing issues; all of which are captured in our models (while we abstract from additional specifics mentioned in [BG05]).

There are several approaches to incorporate some of those specifics of web services into symbolic models. In [BFG05] the applied pi calculus [AF01] is extended to account for specifics of XML encoded data that are used, e. g., by the XML Signature standard. The authors of [CLR07] develop a model that allows non-deterministic receive/send actions as well as unordered sequences of XML nodes, which are used, e. g., in SOAP message headers. Furthermore, a model that faithfully captures SOAP's extendability would have to include open-ended data structures like [CTR09]. An example for attacks that are specific to XML-based protocols is the class of *XML rewriting attacks*, see [BFGO05], which are dealt with, e. g., in [MA05, GLS07, SB08].

Automatic verification tools for symbolic models have also been tailored to the web service setting: The language defined in [BFGP03] allows to specify web-service based protocols and security properties that are verifiable by ProVerif [Bla01], an automatic verifier for cryptographic protocols in the symbolic model. In [BFGT06], a cryptographic library is proposed that offers both a real cryptographic implementation as well as a symbolic abstraction that allows to produce a protocol specification analyzable by ProVerif. In [BFG08], the authors define a specification mechanism for security goals and provide tools to compile those specifications into policy files which are then proven secure against the specified goals using a theorem prover.

Less formal, but more practical approaches include, e. g., [BFGO05], which introduces a tools that analyzes security policies and proposes improvements.

In all the work mentioned above, no concrete protocol for secure message exchange in two rounds has been proposed, formally specified and rigorously analyzed with respect to its security prior to our work. The reasons may lie in the above-mentioned flexibility of the web service standards, but we argue that there exists some prevalent scenarios

(see Section 2.4) for which a concrete protocol should be proposed and analyzed.

We note that we abstract from many details discussed in work cited above, e. g., from XML rewriting attacks, and assume that the processing of XML encoded messages, signatures, and ciphertexts is correctly implemented.

Password-Based Authentication

One of the protocols we propose and analyze partially relies on passwords for authentication. Password-based authentication protocols have widely been analyzed, for example, in [BPR00] referenced above, which extends the Bellare–Rogaway framework, or in [CHK+05] in the Universal Composition framework.

We, however, use an asymmetric setting where we distinguish between servers that can publish keys through a public-key infrastructure (and thus authenticate their messages using digital signatures) and clients which only have passwords to authenticate their messages. In [HK99], several protocols are analyzed in a similar setting.

In [FMCS04], the authors provide a web service based protocol for password-based authenticated key exchange, but neither in two rounds nor in the asymmetric setting.

Random Oracle Model For the analysis of our password-based protocol we use the random oracle model that was proposed in [BR93b] as a paradigm for abstracting from hash functions when analyzing cryptographic protocols.

Despite its serious flaws, for example shown in [CGH04], the random oracle model is used in current work, e. g., in [MSW08, BFCZ08] for the analysis of TLS or in [BP06b] for the analysis of signature schemes.

The random oracle model was also used for analyzing password-based protocols, for example, in [Luc97, BMP00]; nevertheless, there exists work for password-based protocols without resorting to the random oracle model, see, e. g., [SBEW01, GL06].

Other Related Work

The precise meaning of the notion of entity authentication is discussed in [Gol96].

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 models, a long-lived server processes an unbounded number of requests from different clients, which is reminiscent of optimistic contract-signing protocols, where the trusted third party potentially needs to remember an unbounded number of requests, see [ASW98, GJM99]. 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.

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

The payloads for our secure two-round message exchange protocols are determined by the adversary; in [RS09] a framework is proposed that models adversarial input in a general fashion.

In [JKL04], protocols for authenticated key exchange with only two messages (or two rounds in our terminology) are studied, but without payloads nor protection against replay attacks.

Our Contribution

In this thesis and the published work (see notes at the end of the introduction), we specify and analyze protocols for securing a two-round message exchange protocol in a setting inspired by web services, and through this gain insights into notions of authentication as well as the modeling and analysis of cryptographic protocols in different frameworks.

More precisely, we assume that an existing service or protocol consists of exactly two messages by different parties, request and response, and we specify new protocols that view those messages as payload, i. e., wrap each of them up into a new message that is augmented with security-related information. We

1. discuss what *authentication* refers to in a two-round message exchange setting, compared to message authentication and entity authentication,

2. specify concrete and practical protocols for three security goals, namely
   a) signature-authenticated two-round message exchange,
   b) confidential signature-authenticated two-round message exchange, and
   c) password-authenticated two-round message exchange,

   based on what has been discussed in standardization documents, see below,

3. adapt and extend the Bellare–Rogaway framework to the signature-authenticated two-round message exchange setting and prove the first of the three protocols correct and secure in a concrete computational analysis (cf. [KSW10, KSW09a]),

4. model all three security goals in a uniform way in the IITM framework as an ideal functionality (which is parameterized to account for the difference in the three security goals), then model the three proposed protocols in the IITM framework and prove them secure in a simulation-based computational fashion (cf. [KSW09b, KSW09c]),

5. discuss connections and differences between the two frameworks.

A simple protocol for signature-authenticated two-round message exchange works as follows: The client appends a message id (e. g., random nonce or sequence number) to its 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 it 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]; this allows the server to counter replay attacks with only limited long-term memory. Our protocols follow this approach, but use a combination of message id's and timestamps such that we do not have to assume perfectly synchronized clocks.

In addition, we allow that payloads contain parts signed with keys that are also used for signing entire messages, following what web service standards allow or suggest [NKMHB06]. As explained below, the use of the keys for signing parts has to be restricted in some sense, because otherwise no security guarantees would be possible.

At least the first two protocols we propose follow what has been alluded to in various documents. For example, securing messages with timestamps to counter replay attacks has been informally proposed in [NKMHB06]. However, the protocols we present have to the best of our knowledge not been precisely defined nor analyzed previously.

Our first security analysis is based on the seminal work [BR93a] by Bellare and Rogaway. The framework developed therein is, however, not general enough. We extend it in two directions: first, we add digital signatures, local clocks, and timestamps etc., and, second, we add payloads and signature oracles for dealing with signed parts. Our model allows the adversary almost complete control over the local clocks of the principals, the only requirement is that clocks are monotone. In particular, we do neither assume synchronized clocks nor a bounded clock drift. A crucial point in our extension of the Bellare–Rogaway framework is that the latter 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 and the first such notion suitable for two-round protocols.

We then formally define a protocol for the signature-authenticated two-round message exchange setting and prove it secure, given a signature scheme that fulfills some standard security notion.

In contrast to [BR93a], we carry out a concrete computational analysis of the proposed protocol instead of an asymptotic one; we obtain the latter as a consequence.

Our second security analysis of all three proposed protocols is carried out in the Inexhaustible Interactive Turing Machines framework [Kü06a].

First, we define an ideal functionality that is parameterized such that it provides a uniform way to model all three security goals mentioned above. In addition to the secure message transfer itself, the modeling includes expiration of a session on the server side (which is necessary due to the limited memory we assume), an explicit modeling of password guessing in the case of password-based authentication, and a simple, but powerful corruption mechanism. Hence, our modeling is one of the most complex ideal functionalities for protocols we know of.

We then define functionalities that implement the protocols we propose: For each of the three protocols, we give implementations for the client and server parts of the protocol, as well as some additional functionalities that are reused for all three implementations. The three protocols are then shown to securely realize the (parameterized) ideal functionality (in case of the password-based protocol, security holds in the random oracle model).

We also share some insights into the IITM framework gained while modeling the protocols, e. g., on a notion of "correctness" in the IITM framework similar to the one in [BR93a], or the joint-state realization of digital signatures.

After using these two frameworks to analyze our protocol, we discuss the relation between both frameworks. To this end we first show that for a simpler case (mutual authentication protocols), there is a strong connection at least in one direction: mutual authentication protocols that are secure in the sense of the Bellare–Rogaway framework securely realize an ideal mutual authentication functionality in the IITM framework. We then give some insights on the relation for the more complex case of secure two-round message exchange.

In summary, our work provides the first firm theoretical underpinning for realistic *and* secure implementations of services which require authentication and/or confidentiality in two rounds. We gain insights on notions of authentication and define "message exchange authentication" as a new, but natural intermediate step between message and entity authentication. In addition, the analysis serves as a case study for modeling complex protocols in both a concrete computational as well as a simulation-based framework.

Pre-Published Results The first protocol we analyze, SA2ME-1, was specified and analyzed in [KSW10, KSW09a], together with most of the results in Chapter 3. Parts of the results in Chapter 4 were published in [KSW09b, KSW09c], namely the analysis of a simpler modeling of SA2ME-1. In this thesis, we extended the results to uniformly model not only SA2ME-1, but also two other protocols introduced in this thesis. We refer the reader to Section 4.6.5 for remarks on differences between the published work and Chapter 4.

Structure of this Thesis

We introduce the general setting of secure two-round message exchange in Chapter 2. First, after some prerequisites (Section 2.1), we discuss different notions of authentication (Section 2.2). After a general introduction into aspects of securing two-round protocols (Section 2.3), we present three different security goals or protocol classes for securing a two-round message exchange (Section 2.4) and informally introduce the protocols we use to implement a secure two-round message exchange (Section 2.5).

In Chapter 3, we analyze one of these protocols, SA2ME-1, in a model based on the framework for entity authentication introduced by Bellare and Rogaway in [BR93a].

We first introduce both the basic and the adapted model (Sections 3.1 and 3.2), we then formally define SA2ME-1 in this model (Section 3.3), give correctness and security definitions (Section 3.4) for protocols in our model, and finally show that SA2ME-1 is secure and correct according to those definitions (Section 3.5).

In Chapter 4, we use the IITM framework introduced by Küsters in [Kü06a]. After an introduction into the IITM framework (Section 4.1), we propose an ideal functionality that can be parameterized to express the three security goals (Section 4.2). We then give realizations which implement the three protocols (Section 4.3) and prove the security of the given realizations (Section 4.4). We also argue how our realizations can be further implemented (Section 4.5), and then conclude the chapter with some remarks on the IITM framework (Section 4.6).

The relation between both frameworks is studied in Chapter 5. A connection is proven for the case of mutual authentication (Section 5.1), after which we make some remarks for secure two-round message exchange (Section 5.2).

We conclude in Chapter 6.

The Appendices A, B, and C contain pseudo code etc. used and referenced in Chapters 3, 4, and 5, respectively.

2. Secure Two-Round Message Exchange

In this chapter, we introduce the setting of two-round message exchange protocols and their characteristics, and we informally specify three protocols for securing two-round message exchanges.

We start with some prerequisites in Section 2.1 and generally discuss different notions of authenticity in Section 2.2. In Section 2.3, we explain our general approach as to what our modelings include. We then define different security goals for securing two-round message exchanges in Section 2.4 and specify protocols for those goals in Section 2.5.

2.1. Prerequisites

In this section, we introduce terms, notations, conventions, and methods used throughout this thesis.

2.1.1. Two-Round Protocols

Some communication protocols or standards are restricted to just two messages, which are called *request* and *response* throughout this thesis. Usually, one refers to the initiator and thus the sender of the first message as *client*, while the receiver of the first message is called *server*, we also adopt this notion.

The restriction to two rounds is useful, for example, because the server can simply answer the request and then forget about it, i. e., the server does neither have to keep state information for protocol sessions after sending a response (see below for exceptions) nor does it have to care for situations like out-of-order arrival of protocol messages.

For example, the SOAP standard [ML07,NGM$^+$07,KMG$^+$07], which is used as a part of the web services protocol stack, defines "SOAP-Supplied Message Exchange Patterns and Features" (see Section 6 of [KMG$^+$07]). Only two message exchange patterns are defined, "Request-Response" and "SOAP Response" (where the latter simply is a variant of the first in which the request does not conform to the SOAP standard).

Other examples for two-message protocols include remote procedure calls (RPC), see [Sun98], or XML-RPC [Win99], a predecessor of SOAP.

Note that throughout this thesis, we refer to protocols with just two messages as *two-round protocols*, similar to, e. g., [BR93a]; whereas other authors use the term "round" to refer to one message per participant, and thus would call protocols with two messages "one-round protocols", see, for example, [Jou04].

2.1.2. Asymptotic Analyses

Several security definitions and analyses in this thesis are of *asymptotic* nature: Usually, a cryptographic primitive or protocol that is parameterized by a *security parameter* $\eta \in \mathbb{N}$ is called *secure* in an asymptotic analysis if the probability that an adversary can successfully attack the primitive or protocol drops super-polynomially in the security parameter, while the algorithms of the primitive or protocol run in time polynomial in the security parameter. For example, when analyzing encryption schemes, their security is often analyzed asymptotically by viewing the length of the keys as the security parameter.

Formally, the success probability of an adversary against some system that is parameterized by a *security parameter* is then required to be *negligible* in the security parameter, i. e., for every positive polynomial p there exists an integer $n > 0$ such that for all $\eta > n$ the success probability of the adversary against the system run with security parameter η is less than $p(\eta)^{-1}$. Analogously, we say that some probability f is *overwhelming* if $1 - f$ is negligible.

In contrast, our analysis in Chapter 3 of *concrete computational* nature, i. e., we compute precise running times and success probabilities for adversaries.

2.1.3. Cryptographic Primitives

2.1.3.1. Digital Signatures

A *signature scheme* Ω is a triple of algorithms $\Omega = (G, S, V)$ satisfying the following conditions:

1. G is a *key generation algorithm*, i. e., a probabilistic algorithm which expects the security parameter 1^η and produces a pair $(pk^{\text{sig}}, sk^{\text{sig}})$, where pk^{sig} is a *public* (or *verification*) key and sk^{sig} the corresponding *secret* (or signature) key;

2. S is a *signature algorithm*, i. e., a probabilistic algorithm that for any bit string $m \in \{0,1\}^*$ and any secret key sk^{sig} produces a *signature* $S(m, sk^{\text{sig}})$;

3. V is a deterministic *verification algorithm* which on input $((m, S(m, sk^{\text{sig}})), pk^{\text{sig}})$ returns true if $(pk^{\text{sig}}, sk^{\text{sig}})$ has been generated by G.

All algorithms have to run in time polynomial in the sum of the security parameter and the input length. The algorithms are allowed to fail, but for each possible message m, the probability that one of the algorithms fails if a key pair is generated, then used to create a signature of m that is verified afterwards, has to be negligible in the security parameter.

Note that we usually denote public and private keys for signature schemes by pk^{sig} and sk^{sig} with a superscript "sig" to distinguish them from public and private keys for asymmetric encryption schemes, see below.

By $\{m\}_{sk^{\text{sig}}}$ we denote the pair (m, σ) where σ is the output of $S(m, sk^{\text{sig}})$, i. e., $\{m\}_{sk^{\text{sig}}}$ is the bit string m accompanied by a valid signature obtained from the signature scheme

for the signature key sk^{sig}. We call a signature σ a *valid* signature of m for the key pk^{sig} if $V((m, \sigma), pk^{sig})$ returns true.

EUF-CMA Security We recall a standard notion (see [GMR88]) for security of signature schemes, namely *existential unforgeability under chosen message attacks* (EUF-CMA): An adversary against a signature scheme is a probabilistic algorithm that as input receives a public key pk^{sig} generated by the key generation algorithm, and has access to a signature oracle that on input m generates a valid signature of m for the key pk^{sig}. The adversary is successful if it produces a pair (m', σ) with a valid signature σ (for the key pk^{sig}) of some m' which has not been signed by the signature oracle before.

A signature scheme is *EUF-CMA secure* in the asymptotical sense if the success probability of any adversary is negligible in the security parameter.

For a running time t (relative to some machine model, see below), natural numbers q and l, and a probability ε, an 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 in the concrete computational sense if there is no adversary that (t, q, l, ε)-breaks it.

2.1.3.2. Asymmetric Encryption

An *asymmetric encryption scheme* (or *public key encryption scheme*) Σ_{ae} is a triple of algorithms $\Sigma_{ae} = (G, E, D)$, satisfying the following conditions:

1. G is a *key generation algorithm*, i. e., a probabilistic algorithm which expects the security parameter 1^η and produces a pair (pk^{ae}, sk^{ae}), where pk^{ae} is a *public* (or *encryption*) key and sk^{ae} the corresponding *secret* (or *decryption*) key;

2. E is an *encryption algorithm*, i. e., a probabilistic algorithm which for any bit string $m \in \{0, 1\}^*$ (also called *plaintext*) and any public key pk^{ae} produces a *ciphertext* $E(m, pk^{ae})$;

3. D is a deterministic *decryption algorithm* which on input $(E(m, pk^{ae}), sk^{ae})$ returns m if (pk^{sig}, sk^{sig}) has been generated by G.

We denote public and private keys for encryption schemes by pk^{ae} and sk^{ae} with a superscript "ae" (for *a*symmetric *e*ncryption) to distinguish them from public and private keys for signature schemes, see above.

By $\langle m \rangle^{ae}_{pk^{ae}}$ we denote a ciphertext of some plaintext m under a public key pk^{ae}.

IND-CCA2 Security Again, we recall a standard security notion (see [BDPR98]), here *indistinguishability under adaptive chosen-ciphertext attack* (IND-CCA2). An adversary against an asymmetric encryption scheme is a probabilistic algorithm that is handed a public key and has access to a decryption oracle, which on input of a ciphertext y

outputs the corresponding plaintext under the corresponding private key. The adversary has to generate two plaintexts m_0 and m_1 and is then handed the encryption of m_b where b is chosen uniformly at random from $\{0, 1\}$. The adversary is successful if it can correctly determine b without handing m_b to the decryption oracle.[2]

An asymmetric encryption scheme is *IND-CCA2 secure* in the asymptotic sense if the success probability of any adversary is negligible in the security parameter.

2.1.3.3. Symmetric Encryption

A *symmetric encryption scheme* Σ_{se} is a triple of algorithms $\Sigma_{\text{se}} = (G, E, D)$, satisfying the following conditions:

1. G is a *key generation algorithm*, i. e., a probabilistic algorithm which expects the security parameter 1^η and produces a *symmetric* (or *shared*) key k;
2. E is an *encryption algorithm*, i. e., a probabilistic algorithm which for any bit string $m \in \{0, 1\}^*$ (called *plaintext*) and any secret key k produces a *ciphertext* $E(m, k)$;
3. D is a deterministic *decryption algorithm* which on input $(E(m, k), k)$ returns m.

By $\langle m \rangle_k^{\text{se}}$ we denote a ciphertext of some plaintext m under a key k (where se stands for *s*ymmetric *e*ncryption).

IND-CCA2 Security As for asymmetric encryption, we use the notion of *indistinguishability under adaptive chosen-ciphertext attack* for symmetric encryption schemes. As above, the adversary is a probabilistic algorithm that has access to a decryption oracle; but since there is no public key, the adversary also has access to an encryption oracle which, on input of a plaintext x returns the corresponding ciphertext y. Again, the adversary has to generate two plaintexts m_0 and m_1 and later determine b after receiving m_b where b is chosen uniformly at random from $\{0, 1\}$. Analogously to above, a symmetric encryption scheme is IND-CCA2 secure in the asymptotic sense if the success probability of any adversary is negligible in the security parameter.

2.1.3.4. Hybrid Encryption

For large plaintexts, one usually wants to combine the advantages of symmetric and asymmetric encryption, e. g., using the easier key-management of asymmetric encryption while being able to efficiently encrypt large plaintexts.

Therefore, for *hybrid encryption*, one uses an asymmetric encryption scheme Σ_{ae} and a symmetric encryption scheme Σ_{se}, and encrypts a plaintext x under a key pk^{ae} by 1. generating a key k for Σ_{se}, 2. encrypting the plaintext x with scheme Σ_{se} under key k, 3. encrypting the key k with scheme Σ_{ae} under some key pk^{ae}, and 4. transmit both ciphertexts resulting from steps 2 and 3.

For a formal treatment of hybrid encryption, see, e. g., [CS03].

[2] The non-adaptive variant (IND-CCA1), not used in this thesis, restricts the adversary in that it may not use the decryption oracle after receiving m_b.

2.1.3.5. Hash Functions

A *cryptographic hash function* is a function $H\colon \{0,1\}^* \to \{0,1\}^l$ for some $l \in \mathbb{N}$ that—informally—fulfills the following security properties:

- Given a value $y \in \{0,1\}^l$, it should be computationally infeasible to find any $x \in \{0,1\}^*$ with $H(x) = y$ *(preimage resistance)*.
- Given a value $x_0 \in \{0,1\}^*$, it should be computationally infeasible to find x_1 such that $H(x_0) = H(x_1)$ *(second preimage resistance)*.
- It should be computationally infeasible to find $x_0, x_1 \in \{0,1\}^*$ with $x_0 \neq x_1$ but $H(x_0) = H(x_1)$ *(collision resistance)*.

Note that this is an informal definition: For example, for any fixed function h, as $\{0,1\}^l$ is finite, there obviously is a collision (a pair $x_0, x_1 \in \{0,1\}^*$ with $x_0 \neq x_1$ but $H(x_0) = H(x_1)$), and thus, there always exists an adversary which simply outputs a collision. For a more formal treatment of hash functions and the security requirements, we refer the reader to [RS04].

We will later refer to these properties in the context of a random oracle (see below), for which the above security properties hold.

Random Oracle Model When cryptographic hash functions are used as a primitive in protocols, the analysis of those protocols is often carried out in the *random oracle model* to abstract from the complications touched above.

In this model, all parties have access to a random oracle that answers every query with a response chosen uniformly at random from $\{0,1\}^l$ for some fixed $l \in \mathbb{N}$, with the restriction that on subsequent queries with the same input, it answers with the same output. Thus, the random oracle models a randomly chosen function from the domain $\{0,1\}^* \to \{0,1\}^l$ for a fixed length l, but operates as a black-box. Obviously, for a random oracle, the above security properties hold in an asymptotical sense if l is the security parameter.

The random oracle model was proposed as a paradigm for the analysis of cryptographic protocols in [BR93b]. One hopes that the security results obtained in those analyses are a strong evidence that security is guaranteed even if the random oracle is replaced by a concrete hash function that, from experience and research, seems to fulfill the three above-mentioned security properties in practice.

This is not supported by sound reasoning, and as shown in [CGH04], there exist (artificial examples for) signature and encryption schemes that are secure when analyzed in the random oracle model, but that become insecure if the random oracle is replaced by any specific function or even a function chosen uniformly at random from any family of functions. Nevertheless, the random oracle model is used in current work, see related work in Chapter 1.

2.1.3.6. Passwords

A *password* is a sequence of characters that is a shared secret between a system and a user of that system. By generally keeping the password confidential, but including it (in a secure manner) in messages, the server can check if messages that seem to originate from a user are authentic.

Analyzing password-based security systems strongly differs from analyzing other cryptographic primitives like digital signatures, as the concept of a security parameter introduced earlier to analyze security in an asymptotic fashion is unrealistic for passwords:

Passwords are often chosen by humans, who usually choose them such that they are easy to remember or type in. Even if they are generated automatically, one usually tries to limit the length to make them (relatively) easy to type in or even remember. Hence, they are often not chosen randomly, they are usually relatively short (compared to standard sizes for cryptographic keys or nonces etc.) and chosen from a limited set of characters [FH10], and to increase their length, one has to make drawbacks on usability, while key-lengths are "only" bounded by technical limitations.

Therefore, when analyzing password-based protocol in a framework in which security proofs are carried out in an asymptotic sense, we have to assume that the adversary has a non-negligible probability of *success* in some sense, e. g., by simply guessing passwords. Therefore, in Chapter 4, we explicitly model the adversary's abilities, e. g., to guess a user's password. See Section 4.2.2 for some remarks on our modeling of password-based security in that chapter.

2.1.3.7. Trust Models

When using digital means (like digital signatures or passwords) for authentication, one has to associate keys or passwords with real-world objects (persons, organizations, roles etc.). To this end, one must usually use some form of *trust model* that defines which keys or passwords one trusts to correctly authenticate communication partners.

Assume that A wants to verify if messages that claim to belong to party B are authentic. If passwords are used, the situation is usually relatively easy: A can simply keep a (local) list of identities and their passwords (or some information derived from their passwords, like hashed passwords) and check if messages claiming to originate from B contain (information derived from) B's password.

For digital signature keys, A can follow the same approach and manage a list of trusted keys directly. Alternatively, A could use a *public key infrastructure (PKI)*. For example, a *web of trust* approach with some trust metric may be acceptable, e. g., if the authentication requirements are less strict. Otherwise, a *hierarchical PKI* might be useful, where one trusts a limited and fixed set of *certification authorities* that offer a service to bind a B's key to (some aspect of) B's identity by issuing a *certificate* following some trustworthy policy. The last approach is widely used, e. g., in the internet for securing sessions with SSL [HE95] or its successor TLS [DA06].

2.1.3.8. Nonces

Nonces, short for *numbers used once*, are values used at most once for the same purpose. The term usually refers to large random numbers or bit strings, e. g., in the case of an asymptotic analysis, one would use random bit strings from the set $\{0,1\}^\eta$ where η is the security parameter.

If used correctly, nonces have at least two security-relevant applications:
Firstly, if a party randomly choose a nonce, it can assume that the probability that the nonce collides with polynomially many other nonces is negligible, even if those other nonces are not chosen randomly. This is useful, e. g., in challenge-response mechanisms (which we describe in the next section), where a party that sends a challenge wants to be sure that the challenge does not collide with an earlier challenge of any party.

Secondly, one can assume that if a nonce is kept confidential, an adversary that cannot break the confidentiality is not able to guess the nonce and thus cannot break a protocol run. For example, in our protocol PA2ME-1 introduced in Section 2.5.3, we use an encrypted nonce in a setting with a weak form of authentication (passwords) to guarantee that, even if the adversary is able to guess the password, it cannot break into a session of the protocol by simply faking a request message, as it does not know the nonce that is used by the client (but it can start new sessions, see below).

2.2. Notions of Authentication

In the introduction to [BR05], Bellare and Rogaway informally introduce *authenticity* (in a network setting) in the following way:

> We want the receiver, upon receiving a communication pertaining to be from the sender, to have a way of assuring itself that it really did originate with the sender, and was not sent by the adversary, or modified en route from the sender to the receiver.

This is only a rough definition, from which several more precise security goals arise (in [BR05] and throughout the literature); two common goals include *message authentication* and *entity authentication*: Informally, message authentication expresses that if a party B receives a messages that appears to originate from a sender A, the receiver B knows that at some point in the past, A has seen the message and actively approved it[3]; but B has no guarantee when this happened. In contrast, entity authentication is usually defined in the context of protocols, where A and B exchange multiple messages: Here, B wants to be sure that it is "talking to A", i. e., that A receives and sends the messages that B sends and receives at this time, respectively.

However, protocols that provide entity authentication usually use more than two rounds to fulfill their security goal(s): In [BR93a, Section 4.2], Bellare and Rogaway give

[3]Here, it is important to distinguish between the payload and the message itself, i. e., the payload may originate at another party, but A has to add some security measures (e. g., a digital signature) to the message such that the message as a whole originated from A.

the following definition for authentication protocols, where "mutual authentication" stands for "mutual entity authentication":

> DEFINITION. We require that any mutual authentication protocol have $R \geq 3$ rounds. We implicitly make this assumption in our definition and throughout the remainder of this paper. [...]

Thus, the question arises what form of authentication is possible if communication is restricted to two rounds. As shown below, the goal of mutual entity authentication in only two rounds requires additional assumptions like synchronized clocks. But on the other hand, as shown in this thesis, we can achieve more than message authentication with simple assumptions like the existence of monotonous clocks.

Figure 2.1 gives an informal overview (which is not intended to be exhaustive) of which authentication goals can be reached by what means (For some further notes on the interpretation of authentication goals, we refer the reader to [Gol96].):

- As noted above, *message authentication* only guarantees the *authenticity* of the message itself, i. e., if it was really emitted by the alleged sender. This goal can be reached using digital signatures if one assumes that 1. a party may certify its identity by demonstrating the knowledge of a certain secret key and 2. there is some mechanism to link keys to identities, see Section 2.1.3.7.
- *Message exchange authentication* informally adds *freshness*, i. e., guarantees that a message was never received and accepted before[4]. As shown below, adding nonces and timestamps to message authentication is one possibility to achieve this security goal under reasonable assumptions.
- Finally, *entity authentication* adds *synchronicity*, i. e., guarantees that in an exchange of messages, the partner is running the protocol at the same time. This can be achieved by adding the assumption of synchronous clocks to the message exchange authentication setting, or alternatively by a challenge–response mechanism (see below).

For illustrating synchronicity, assume that in a protocol, party A randomly chooses a nonce r, sends it to party B, and later receives a message m signed by B that contains r. Then, if the set from that r is chosen is large enough to prevent guessing and we assume that only B has the power to sign messages with its private key, A can at least be sure that B was active *after* A chose r. This may give A the guarantee that A and B are active "at the same time". The general mechanism behind this is referred to as *challenge–response*.

But in two-round protocols, the receiver of the first message, the server, can only send one message, hence, it cannot send a challenge to the client, as it cannot receive a response. Thus, from the message alone (without further assumptions), the server

[4] Again, it is important to distinguish between the payload and the message itself, i. e., the same payload may be received multiple times, but only if it was sent multiple times in different messages.

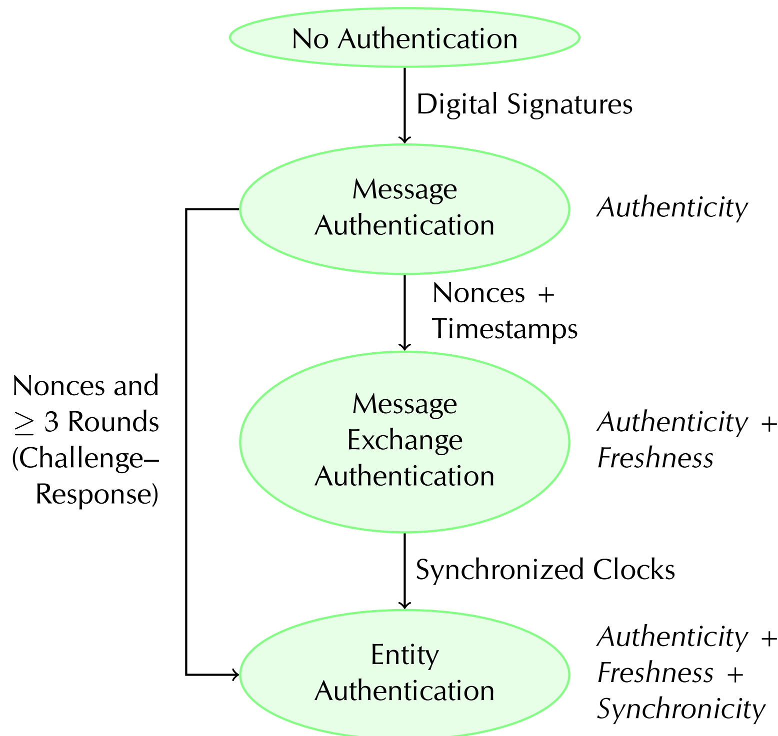

Figure 2.1.: Authentication means and goals

cannot know if both client and server are active "at the same time" in the sense above; the client may have created the request message a long time ago (which may not be detectable due to asynchronous clocks) and the message may have been delayed, e. g., due to network failure or an attack.

The challenge–response mechanism would also allow the server to resist *replay attacks:* A replay attack occurs if an adversary that has some control over the network and overhears the communication between two parties later replays the communication made by one party to trick the other party into believing that another session of the protocol is running.

In an unprotected two-round protocol with client and server as above, the attacker may for example be able to just send two copies of a client's message to a server, making the server believe that the client has made two (equal) requests etc. If the server would be allowed to send more than the response message, it could for example send two different challenges back and wait for two (different) responses. But in two rounds, another mechanism is necessary to prevent replay attacks and guarantee freshness as defined above.

Therefore, we use timestamps and equip the server with long-term memory as explained in Section 2.3.2.

2.3. Securing Two-Round Message Exchange

2.3.1. Message Wrapping and Alternative Approaches

In this thesis, we secure protocols that consist of two messages. To this end, we secure each of those messages by wrapping them in a new message that also contains some security-related parts. The original messages are referred to as *payloads* throughout this thesis.

There is an alternative: Protocols like SSL [HE95], TLS [DA06], or SSH [Ylö96] could be used to secure a message exchange protocol by first establishing a secure connection between two parties and then running the desired message exchange protocol over this connection. This increases the number of messages that have to be exchanged between the two parties, hence, we would not refer to such an approach as secure two-round message exchange. See Section 2.3.1.1 below for some further notes on why we do not choose this approach.

To be precise, there is another approach that is used in the WS-Security standard to secure SOAP messages [NKMHB06]: A SOAP message already is divided into a *SOAP header* and a *SOAP body*, where the SOAP body contains the payload that SOAP is supposed to encode and transfer, while the flexibly extendible SOAP header contains metadata like sender and receiver of the message etc.

WS-Security then defines some additional elements that can be inserted into a SOAP header to encode security-related information such as signatures and timestamps. In addition, one also uses header elements defined elsewhere, e. g., *MessageID* and *RelatesTo* defined in WS-Addressing [BCC+04], to further secure messages.

Thus, a SOAP message is secured not by wrapping the message into a new message, but by inserting the appropriate elements in the SOAP header, which has a pre-defined flexible structure that allows for such insertions.

We remark that this third possibility is only syntactically different from our approach, where a message is viewed as payload and then wrapped into a new message; all the results presented in this thesis would also hold if we would take the approach of WS-Security. But the third possibility is less flexible in that it is restricted to messages which allow for such insertions, therefore, we use the message wrapping approach.

Note that XML Signatures offers means to sign multiple elements (like, e. g., the message body together with a list of headers) in one signature, binding those parts of the message together; thus, even if in our protocols messages are signed as a whole, this can be transferred to the encoding of WS-Security where the parts that have to be signed are scattered throughout the SOAP message.

2.3.1.1. Connection-Centric and Document-Centric Views

As explained above, a common way to secure a protocol is to run the entire connection for that protocol over a special security protocol. For example, a web service message may be sent over an HTTPS connection, i. e., a HTTP connection that is secured by

TLS [DA06]; this may allow the receiver to check the authenticity of the message and ensure confidentiality during the transmission etc. Similarly, the connection could be secured using SSH [Ylö96] etc.

But there are drawbacks to this approach: First, the data transferred is only secured during the transmission; so, after a message is transferred, the data is no longer secured on the receiver's side (e. g., while being stored or processed). This is undesirable if it is necessary to, e. g., ensure confidentiality not only during transmission, but also during further processing or storage at the receiver's side.

Second, the protocol's parties may not be able to establish a direct connection. Instead, they may, for example, transfer messages over intermediary systems that, e. g., cache several messages or preprocess parts of the messages and thus play an active role in the protocol. Then, each of these connections between one system and the next one may be secured using TLS or similar measures, but this does not ensure that the message is secured while stored or processed on an intermediary system.

Therefore, in the world of web services, one often has takes *document-centric view*, where messages are viewed as stand-alone documents, which should be secured as such, in contrast to a *connection-centric view* that focuses on only secures the connections between parties. In [NKMHB06, Section 1.1.1], security for the document-centric view is referred to as "end-to-end message content security" in contrast to transport-level security.

Hence, we provide securement on the document or message level. This does not rule out combining our approach with, e. g., TLS: In our analysis, we show that security is guaranteed independently from the underlying transport scheme, hence, adding another security layer is no drawback, but might be useful, e. g., to reach additional security goals: Running our SA2ME-1 protocol (which offers no confidentiality) over a connection that is secured by TLS may result in both authenticity of the message as well as confidentiality during the transmission.

2.3.2. Features of our Models

2.3.2.1. Timestamps

To overcome the above-mentioned problem of the server to check if a client is active "at the same time", one can include timestamps in the messages. But if one relies on these timestamps, one would also have to assume that all parties have access to synchronized clocks. Besides the fact that hardware clocks are seldomly synchronized perfectly, there may be situations where an adversary may be able to manipulate a party's clock, cf. [NKMHB06, Section 13.1].[5] Thus we assume that the principals have access to local, but not synchronized clocks.

This also allows the server to be flexible in the sense that it may choose to on the one hand trust the timestamps in the messages, but on the other hand explicitly accept old

[5] A simple example includes clocks that are regularly synchronized with time servers, which's response can be manipulated if the adversary has control over the network.

messages, for example, to account for caching or processing of intermediary parties as introduced in Section 2.3.1.1.

Note that naturally, the protocols we analyze are limited in the sense that if the asynchronism between the clocks of the protocols' parties is "too large", messages may get rejected. But by making the assumption that the principals' clocks are not synchronized at all, we can show that no security threat arises from any possible asynchronism.

Also note that later on, we make the assumption that the clocks of the principals are at least monotonous, i. e., the adversary is only able to increase the value of a principal's clock, not decrease it. In Section 3.5.2.1, some insights are given as to why this assumption is necessary and how one could even restrict the assumption to principals which act as a server.

2.3.2.2. Long-Term, but Limited Memory

To protect against replay attacks, it seems necessary to assume that the server has access to long-term memory. This allows a simple security mechanism:

Each client that wants to initiate a protocol session randomly chooses a *message id* that serves as a nonce and that is included in the request message. The server then stores all message id's it receives and thus can reject messages that arrive more than once.

But in this simple way, the server would have to store all message id's it has ever received, which is highly impractical. Therefore, we model the restriction that the server should not be forced to store all message id's by limiting the memory available to the servers. To this end, we later introduce the *capacity* of a server.

Our protocols then describe how to store as many message id's as the capacity allows, but not more, by using the timestamps included the messages to further restrict the set of messages that a server accepts at any given time.

2.3.2.3. Shared Access to Signature Keys

We assume that we do not have exclusive access to the keys used for digital signatures in our protocols, but that instead access to the keys is shared, e. g., by different applications on one machine.

This has the following background: A signature key that is used to authenticate the message of our protocols may also be used, e. g., to authenticate parts of the payloads. For example, if a request contains parts that the server does not process itself, but that are forwarded to third parties, these parts may also have to be authenticated.[6] As managing multiple keys may be complicated or expensive, the client should be able to sign parts of the payload with the same signature key that is also used to sign the whole

[6] A simple example is a request message that orders some goods from a merchant and that also contains a part that instructs a bank to transfer money from the client's account to the merchant. Both the order itself as well as the payment instructions should be authenticated, and the merchant should be able to extract the authenticated payment instructions from the message. Now, signing the payment instruction and then including the signed payment instruction in the order message, which itself gets signed as a whole later on, allows the merchant to extract the signed payment instructions.

message. The same may apply to servers that use their keys to not only sign messages of our protocols, but multiple protocols.

We model this shared access by providing access to the signature keys used to sign messages, but the access is restricted in that it is not allowed to sign bit strings that represent protocol messages. Otherwise, no security could be guaranteed.

See [BG05] for some further notes on shared access to signature keys.

2.3.2.4. Asymmetric Access to Methods for Authentication

In addition to two protocols using digital signatures for authentication, we also analyze a protocol that for authentication partially relies on passwords instead of digital signatures: We assume that servers use digital signatures to authenticate their messages, whereas clients use passwords for authentication, the reasons are explained in this section.

While digital signature schemes provide a high level of security, they also come with a price: For clients that want to use a service on the internet offered by some server, generating and managing a signature key pair is more complicated than choosing and managing a password. For example, one has to securely store the private key, making it less mobile than a password, which can simply be entered at any machine. In addition, it is usually not feasible for clients to go through the (costly and technically demanding) process of obtaining a certificate for a public-key infrastructure to prove the client's identity.

In contrast, a password is easy to securely transfer using secondary media—for example, the operator of a server can send letters to its users containing (short) printed passwords (which are easier to type for users than long cryptographic keys). From this and other reasons, many online services let their clients authenticate themselves using passwords.

For servers, the situation is different: If a client connects to a server and expects it to authenticate itself, then the client often does not want to manually manage a list of trusted servers or even manually check the identity of a server.

Instead, one usually expect servers to authenticate themselves using means that can be easily checked automatically by any user. As the operator of a server is usually able to obtain a certificate in a public-key infrastructure, e. g., for its domain name; this is a widely-used approach for online services.

2.4. Protocol Classes

We now introduce three different classes of protocols, i. e., we informally define three different security goals for secure two-round protocols:

1. Signature-Authenticated Two-Round Message Exchange (SA2ME),
2. Confidential Signature-Authenticated Two-Round Message Exchange (CSA2ME),

3. Password-Authenticated Two-Round Message Exchange (PA2ME).

We use these informally defined protocol classes throughout the thesis, and we refer to the set of all three classes by *Secure Two-Round Message Exchange (S2ME)*.

We remark at this point that it would make sense to define a fourth class, CPA2ME, which combines the attributes of confidentiality as in CSA2ME and password-based authentication as in PA2ME. A protocol for this class would be useful in practice in situations where both confidentiality and password-based authentication are necessary (the latter, e. g., due to reasons mentioned in Section 2.3.2.4).

We do not analyze this class nor propose a protocol in this work, as we do not think that this would give new insights into aspects that are not already mentioned when analyzing CSA2ME and PA2ME. Nevertheless, we believe that a protocol for this class is possible along the lines of the protocols proposed below for CSA2ME and PA2ME, and we note that our ideal functionality proposed in Section 4.2 is parameterized in a way that would allow us to capture this class.

2.4.1. Signature-Authenticated Two-Round Message Exchange

The first class, *Signature-Authenticated Two-Round Message Exchange (SA2ME)*, achieves authentication by using digital signatures. We assume that all parties, i. e., clients and servers, have access to signature keys as well as to a public key infrastructure etc. that allows them to check authenticity of a message based on a signature on that message.

This is the protocol class introduced and analyzed in [KSW10,KSW09a] and also analyzed in [KSW09b,KSW09c], where we refer to this class as “Two-Round Authenticated Message Exchange”, or 2AMEX in short.

2.4.2. Confidential Signature-Authenticated Two-Round Message Exchange

The second class is called *Confidential Signature-Authenticated Two-Round Message Exchange (CSA2ME)* and achieves authentication by using digital signatures as in the first class, but adds confidentiality of both the request and the response payload as a security goal. We assume that clients have access to keys of the servers for an asymmetric encryption scheme, for example, through the same public key infrastructure that is also used for the signature keys.

We stress that only the payload is kept confidential and not, for example, the sender or receiver of the message (which would, e. g., lead to problems when routing messages). In addition, the protocols usually still leak information about the payloads, e. g., their lengths; see Section 4.3.1.4.

2.4.3. Password-Authenticated Two-Round Message Exchange

The third class, *Password-Authenticated Two-Round Message Exchange (PA2ME)*, achieves authentication by using passwords for authenticating clients to the servers and digi-

tal signatures for authenticating servers to the clients; this asymmetry is explained in Section 2.3.2.4. Thus, we assume that

- clients have a password (possibly different one for each server they want to connect to),
- servers have a list of clients and their passwords,
- servers have signature keys, and
- clients have access to a public key infrastructure that allows them to check authenticity of a message received from a server based on a signature on that message.

Naturally, the security guarantees for (short, memorable) passwords are weaker than for signatures-based authentication, see Sections 2.1.3.6 and 4.2.2 for some remarks. Nevertheless, password-based authentication is widely used in practice.

2.5. Proposed Protocols

In this section, we informally describe our protocols called SA2ME-1, CSA2ME-1, and PA2ME-1 for the three above-mentioned protocol classes.

2.5.1. Signature-Authenticated Two-Round Message Exchange

In SA2ME-1, a protocol for the SA2ME protocol class, a signature-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 $\{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c^{\mathrm{sig}}}$ 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 $\{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)\}_{sk_s^{\mathrm{sig}}}$ 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 id 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. 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 $t \notin [t_{\min} + 1, t_s + \mathrm{tol}_s^+]$, 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.

Note that this is the protocol introduced in [KSW10, KSW09a] and also analyzed in [KSW09b, KSW09c], where we referred to this protocol as *2AMEX-1*.

We also note that in Chapter 4, we slightly modify the protocol to allow for a more uniform modeling (compare CSA2ME and PA2ME below): The server does not pass on r to the service, but a randomly chosen session id sid_s, which is also stored in L together with t, r, and c; see Section 4.6.5 for some further remarks.

2.5.2. Confidential Signature-Authenticated Two-Round Message Exchange

CSA2ME-1 is a protocol for the protocol class CSA2ME. A confidential and signature-authenticated message exchange between a client c and a server s works as follows, where most of the variables are the same as for SA2ME-1 (we point out the differences below).

1. a) c is asked by a user to send the request p_c.
   b) c sends $\{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Key}\colon \langle k \rangle^{\mathrm{ae}}_{pk_s^{\mathrm{ae}}}, \mathsf{Body}\colon \langle p_c \rangle^{\mathrm{se}}_k)\}_{sk_c^{\mathrm{sig}}}$ to s.
   c) s checks whether the message is admissible and if not, stops.
   d) s forwards the request (sid_s, p_c) to the addressed service.

2. a) s receives a response (sid_s, p_s) from the service.
   b) s checks whether the response is admissible and if not, stops.
   c) s sends $\{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon \langle p_s \rangle^{\mathrm{se}}_k)\}_{sk_s^{\mathrm{sig}}}$ to c.
   d) c checks whether the message is admissible and if not, stops.
   e) c forwards the response p_s to the user.

Most of the variables are the same as for SA2ME-1. Here, sid_s is a randomly chosen nonce and k is a randomly generated key for the symmetric encryption scheme; thus, we use hybrid encryption as explained in Section 2.1.3.4 to encrypt the request payload. The response is encrypted using the same key k as for the request. The processing is as above, but in addition, sid_s and k are stored in the set L.

2.5.3. Password-Authenticated Two-Round Message Exchange

For the protocol class PA2ME, we informally define the protocol PA2ME-1, in which a password-authenticated message exchange between client c and server s runs as follows:

1. a) c is asked by a user to send the request p_c using password pw.
   b) c computes $m'_c = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon H(r), \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)$ and sends $m_c = (m'_c, \langle(\mathsf{SecMsgID}\colon r, \mathsf{Pass}\colon pw, \mathsf{MsgHash}\colon H(m'_c))\rangle^{\mathrm{ae}}_{pk^{\mathrm{ae}}_s})$ to s.
   c) s checks whether the message is admissible and if not, stops.
   d) s forwards the request (sid_s, p_c) to the addressed service.

2. a) s receives a response (sid_s, p_s) from the service.
   b) s checks whether the response is admissible and if not, stops.
   c) s sends $\{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon H(r), \mathsf{Body}\colon p_s)\}_{sk^{\mathrm{sig}}_s}$ to c.
   d) c checks whether the message is admissible and if not, stops.
   e) c forwards the response p_s to the user.

Again, most of the variables and processing is as above. Here, H is a cryptographic hash function.

The client does not chose a random message id, but instead randomly chooses a *secret message id* r and uses the hash value $H(r)$ as the (public) message id. The request is not signed by the client; instead, a token containing the hash value of the message, the client's password, and the secret message id r is encrypted using the public key of the server.

Note that instead of the password we could also send a hash value of the password etc. if the password verification mechanism on the server allows this.

3. Trace-Based Analysis

The first framework that we use to analyze one of the three protocols, SA2ME-1, is an extension and adaptation of the framework for entity authentication introduced by Bellare and Rogaway in [BR93a]. In this chapter, we adapt the original framework and then perform a concrete computational security analysis of SA2ME-1 in our adapted model.

In Section 3.1, we briefly introduce the original Bellare–Rogaway framework [BR93a] for reference. Then we introduce our model for analyzing SA2ME protocols in Section 3.2, after which, in Section 3.3, we are able to formally define SA2ME-1 as a protocol in our model. We then give correctness and security definitions, see Section 3.4, and prove SA2ME-1 secure and correct in Section 3.5. Finally, we end the chapter with some practical considerations on the choice of parameters in Section 3.6.

Most of the results in this chapter were published in [KSW10, KSW09a].

3.1. The Bellare–Rogaway Framework

In this section, we give a brief introduction into the framework proposed by Bellare and Rogaway in [BR93a], which provides a way to formally specify protocols for entity authentication and key distribution and prove their security and correctness.

In this framework, all communication between the parties is under the control of an adversary, i. e., the adversary has control over an unbounded number of *sessions*, to which it can send messages and from which it receives all of their outgoing messages. A session is denoted with $\Pi_{i,j}^s$, where i is the party running the session with local session id s, and j is the intended communication partner.

For each session, the communication history is recorded in a *trace*, which is the sequence of incoming and outgoing messages. Informally, two sessions $\Pi_{i,j}^s$ and $\Pi_{j,i}^t$ have *matching conversations* if the messages sent by each one are exactly those received by the other (except that the final message is allowed to get lost, which models that the sender of the last message in a protocol can never be certain whether the message is received). A protocol is regarded to be *secure* if the only way that an adversary can get a party to accept the run of a protocol is by faithfully relaying messages.

Protocols. The protocols in this framework are specified by an efficiently computable function Π, which is called each time that a principal i receives a message m_{in} and is supposed to respond to that message. The function has the following input parameters: the security parameter 1^η, the identity of the sender and the receiver $i, j \in I \subseteq \{0,1\}^\eta$,

the private information of the sender $a \in \{0,1\}^*$, the message trace $\kappa \in \{0,1\}^*$ (including m_{in}), and a random bit string $r \in \{0,1\}^\infty$.

In the above, I is a set of identities which define the players of the protocol, while the identity of the adversary, $\mathcal{A}$, is not contained in this set. The private information a is generated by a long-lived key generator $\mathcal{G}$, described below. The message trace κ represents the sequence of messages transmitted and received by i so far in this run of the protocol, and m_{in} is appended to that set before calling Π.

The return value of $\Pi(1^\eta, i, j, a, \kappa, r)$ is a tuple $(m_{\text{out}}, \delta, \alpha)$ with the response message $m_{\text{out}} \in \{0,1\}^* \cup \{\bot\}$, the decision $\delta \in \{\mathsf{A}, \mathsf{R}, *\}$, and the private output $\alpha \in \{0,1\}^* \cup \{\bot\}$ (see below). Here, m_{out} is the message to send out, where $\bot$ stands for "no message". There are three possible values for the decision δ: A for "accept" or R for "reject" denote that the principal believes that the authentication has been successfully completed or that is has failed, respectively; finally $*$ denotes that the principal has not reached a decision yet. It is convenient to assume that as soon as $\delta = \mathsf{A}$, the values of δ and α do not change for subsequent calls of the protocol.

The LL-Key Generator $\mathcal{G}$. The long-lived key generator $\mathcal{G}$ is a polynomial-time algorithm that computes the private information of the parties, which may for instance be used for distributing a shared key or modeling public information accessible by all parties. As input, the algorithm takes the security parameter 1^η, the identity of a party or the adversary ($i \in I \cup \{\mathcal{A}\}$), and a random bit string $r_{\mathcal{G}} \in \{0,1\}^\infty$. The output of $\mathcal{G}(1^\eta, i, r_G)$ is handed to party i.

The Experiment. To define the *success* of an adversary, an *experiment* is used that models a typical protocol execution. In a setup phase, random bit strings $r^s_{i,j}$ are chosen for all sessions, the long-lived key generator is used to generate private information a_i for all identities $i \in I \cup \{\mathcal{A}\}$, and the private information $a_{\mathcal{A}}$ is given to the adversary.

The adversary may now ask queries of the form (i, j, s, m_{in}) to a session $\Pi^s_{i,j}$. Then, m_{in} is appended to $\kappa^s_{i,j}$, the protocol computes $(m_{\text{out}}, \delta, \alpha) = \Pi(1^\eta, i, j, a_i, \kappa^s_{i,j}, r^s_{i,j})$, and (m_{out}, δ) is output to the adversary. Note that the session id s is not given to Π in the input parameters, i. e., the protocol itself has to use some mechanism to distinguish different runs of the protocol. The experiment is run until the adversary terminates.

Matching Conversations. After the run of the experiment, the conversation that the session $\Pi^s_{i,j}$ had with the adversary is denoted by $K^s_{i,j}$, which is a sequence of tuples of the form $(\tau, m_{\text{in}}, m_{\text{out}})$ denoting that i received a message m_{in} and responded with m_{out} at step τ. For a protocol with $2n+1$ messages for some $n \in \mathbb{N}$ the session $\Pi^s_{i,j}$ has a *matching conversation* with $\Pi^t_{j,i}$ if there exists $\tau_0 < \tau_1 < \ldots < \tau_{2n}$ such that

$$K^s_{i,j} \text{ starts with } (\tau_0, \varepsilon, m_0), (\tau_2, m_1, m_2), \ldots, (\tau_{2n}, m_{2n-1}, m_{2n}) \text{ and} \tag{3.1}$$

$$K^t_{j,i} \text{ starts with } (\tau_1, m_0, m_1), (\tau_3, m_2, m_3), \ldots, (\tau_{2n-1}, m_{2n-2}, m_{2n-1}) \tag{3.2}$$

The other direction is similar: $\Pi^t_{j,i}$ has a *matching conversation* with $\Pi^s_{i,j}$ if there exists $\tau_0 < \tau_1 < \ldots < \tau_{2n}$ such that $K^s_{i,j}$ starts as in (3.1) and

$$K^t_{j,i} \text{ starts with } (\tau_1, m_0, m_1), (\tau_3, m_2, m_3), \ldots, (\tau_{2n+1}, m_{2n}, *) \ . \tag{3.3}$$

The case for protocols with an even number of messages (greater than two) is analogously.

Mutual Authentication and Authenticated Key Exchange. A protocol Π is a *correct* and *secure mutual authentication protocol* if for any polynomial-time adversary the following properties hold:

1. If any two sessions $\Pi^s_{i,j}$ and $\Pi^t_{j,i}$ have matching conversations, then both sessions accept *(correctness)*.
2. The probability that a session $\Pi^s_{i,j}$ accepts without having a matching conversation with some session $\Pi^t_{j,i}$ is negligible *(security)*.

The case of authenticated key exchange extends the above-mentioned mutual authentication by using the private output α of Π as the key that is generated or exchanged in this run of the protocol. Therefore, if two sessions $\Pi^s_{i,j}$ and $\Pi^t_{j,i}$ have matching conversations, they should output the same value α, but the adversary should not be able to get any information about α. A formal definition of correctness and security of such protocols can be found in [BR93a].

Note that Shoup [Sho99] corrects a serious flaw in the authenticated key exchange modeling of [BR93a], but this is not relevant for what follows as that is based on the modeling of mutual authentication in [BR93a].

3.2. Protocol Model

In this section we define the protocol model that is later used to analyze SA2ME protocols and that is based on the framework introduced in the previous section. In some footnotes, we point out differences and similarities to the model in Chapter 4.

3.2.1. Prerequisites

As shown in Section 2.3.2.1, in a bounded memory setting time is a way to achieve resistance against replay attacks. In this chapter, we use l_{time}-bit numbers as time values for an arbitrary fixed $l_{\text{time}} \in \mathbb{N}$. In this chapter, we also assume there is an arbitrary fixed *identifier set* $\text{IDs} \subseteq \{0,1\}^{l_{\text{ID}}}$ 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. For the remainder of this chapter, we assume a fixed signature scheme $\Omega = (G, S, V)$ as defined in Section 2.1.3.1.

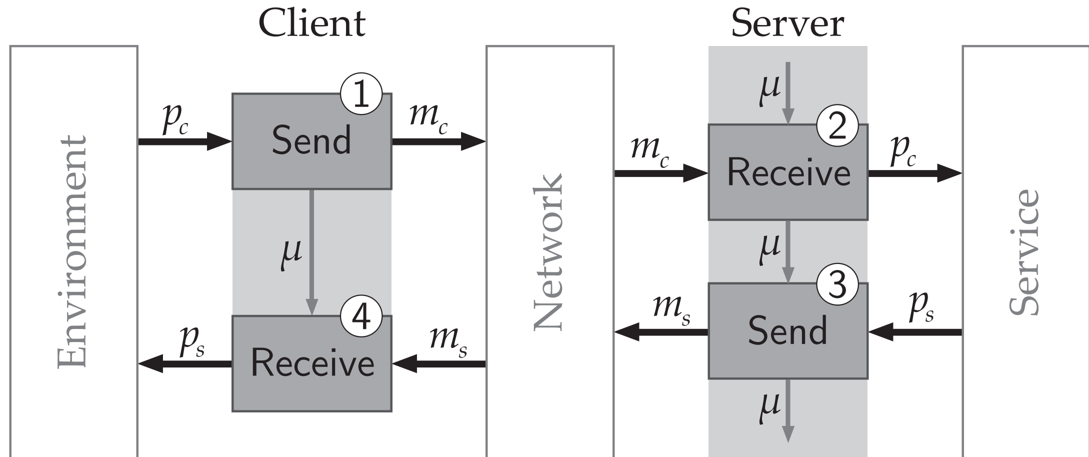

Figure 3.1.: Message flow in four steps

3.2.2. Clients and Servers

Before defining clients and servers formally, we describe how they are supposed to operate. An intended run of a SA2ME protocol between a client $c \in \text{IDs}$ and a server $s \in \text{IDs}$ is initiated by the client-side *environment* which wants to call some *service* on the server.[7] The protocol run consists of two rounds, request and response, modeled by four steps as illustrated in Figure 3.1:

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; passing 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 this model[8]. As the adversary is free to choose any payload both on the client as on the server side, the security

[7]Note that in this chapter, we distinguish between the *environment* on the client side and the *service* on the server side, whereas in Chapter 4, both roles are played by one machine called *environment*.
[8]Note that in Chapter 4, these three parties are also able to work together.

input parameters	client Γ	server Σ
instruction	$\alpha \in \{\mathsf{Send}, \mathsf{Receive}\}$	$\alpha \in \{\mathsf{Send}, \mathsf{Receive}, \mathsf{Reset}\}$
identity	$c \in \mathrm{IDs}$	$s \in \mathrm{IDs}$
partner's identity	$s \in \mathrm{IDs}$	
public keys	$pk^{\mathrm{sig}}_{\mathrm{IDs}}$	$pk^{\mathrm{sig}}_{\mathrm{IDs}}$
private key	sk^{sig}_{c}	sk^{sig}_{s}
local time	$t \in \{0,1\}^{l_{\mathrm{time}}}$	$t \in \{0,1\}^{l_{\mathrm{time}}}$
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 values	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		$c \in \mathrm{IDs} \cup \{\varepsilon\}$
message handle		$h \in \{0,1\}^*$
local state	μ'	μ'

Table 3.1.: Input parameters and output values of the algorithms Γ and Σ

guarantees we provide in this chapter apply to any protocol or service which uses any secure SA2ME protocol.[9]

Formally, we define *client* and *server algorithms* to be probabilistic algorithms with input parameters and output values as specified in Table 3.1 (and explained in the next section) and adhering to the restrictions defined in Section 3.2.2.2.

3.2.2.1. Input Parameters and Output Values

First, the client algorithm gets an instruction which can either be Send or Receive. Second, the client algorithm is provided with the identifier of the principal it is running for, $c \in \mathrm{IDs}$, and with the identifier of the server it is supposed to be talking to, $s \in \mathrm{IDs}$. Third, the client algorithm is provided with the family of public keys, $pk^{\mathrm{sig}}_{\mathrm{IDs}} = \{pk^{\mathrm{sig}}_{a}\}_{a \in \mathrm{IDs}}$, and its own private key sk^{sig}_{c}. Fourth, it gets the local time $t \in \{0,1\}^{l_{\mathrm{time}}}$. Fifth, the client is provided with the payload $p \in \{0,1\}^*$ it is supposed to send to the server, or with a message $m \in \{0,1\}^*$ obtained from the network. Finally, a client may process 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 the client algorithm is provided with—the parameter $\mu \in \{0,1\}^*$, initialized with ε.

For the server algorithm, 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

[9] This is a very simple form of composability, whereas in the framework used in Chapter 4, composability is formally defined.

has to extract this from the message it receives and store it in its memory to be able 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 which message the service wants to respond to. Also, there is an additional Reset instruction for the server, where the local state and the message given to the server is ε.

Next, we explain the output values for the server when receiving a message m_c. First, the server algorithm extracts the payload $p \in \{0,1\}^*$ carried by m_c and returns it. Second, the server algorithm reports its *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 outputs the identity of the client that assumably sent the message; when the decision is R, the dummy value ε is used. Fourth, the server outputs a message handle $h \in \{0,1\}^*$ which is later used by the service to respond to m_c. Finally, the server outputs local state, which it is provided with the next time it is called, unless it is reset.

If the server is asked to send a response payload p_s, the output syntax is similar, but in case no error occurs when responding (see below), the server outputs the response message m_s in the first component, the identity of the receiver in the third component, and ε instead of a message handle in the fourth component.

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 parameters of the algorithm.

3.2.2.2. 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 a monotonically increasing sequence of timestamps and $\{b_j\}_{j\in\mathbb{N}}$ a sequence of bit strings. Assume that for all $i \in \mathbb{N}$ we have

$$\Gamma(\alpha_i, c, s, pk^{\mathrm{sig}}_{\mathrm{IDs}}, sk^{\mathrm{sig}}_c, t_i, b_i, \mu_i) = (b'_i, \delta_i, \mu_{i+1}) \ , \tag{3.4}$$

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, let $\{\alpha_j\}_{j\in\mathbb{N}}$ with $\alpha_j \in \{\mathsf{Send}, \mathsf{Receive}, \mathsf{Reset}\}$ be a sequence of instruc-

tions, and let $\mu'_{-1} = \varepsilon$. If for all $i \in \mathbb{N}$ we have

$$\Sigma(\alpha_i, s, pk^{\text{sig}}_{\text{IDs}}, sk^{\text{sig}}_s, t_i, b_i, h_i, \mu_i) = (b'_i, \delta_i, c_i, h'_i, \mu'_i) \tag{3.5}$$

$$\text{with } \mu_i = \begin{cases} \varepsilon & \text{if } \alpha_i = \text{Reset} \\ \mu'_{i-1} & \text{otherwise,} \end{cases} \tag{3.6}$$

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

3.2.2.3. Message Equivalence

We now define an equivalence relation $\equiv$ on messages: Intuitively, two messages are equivalent if the output of client and server algorithms is the same no matter which of the two messages we give to the algorithm. This relation is used in the security definition below.

Formally, fix a client and a server algorithm, Γ and Σ, respectively. Two bit strings b_1, b_2 are equivalent, denoted by $b_1 \equiv b_2$, if the following conditions hold for all instructions $\alpha \in \{\text{Send}, \text{Receive}\}$, identifiers c, s, timestamp t, and bit strings μ and h:

$$\Gamma(\alpha, c, s, pk^{\text{sig}}_{\text{IDs}}, sk^{\text{sig}}_c, t, b_1, \mu) = \Gamma(\alpha, c, s, pk^{\text{sig}}_{\text{IDs}}, sk^{\text{sig}}_c, t, b_2, \mu)\ , \tag{3.7}$$

$$\Sigma(\alpha, s, pk^{\text{sig}}_{\text{IDs}}, sk^{\text{sig}}_s, t, b_1, h, \mu) = \Sigma(\alpha, s, pk^{\text{sig}}_{\text{IDs}}, sk^{\text{sig}}_s, t, b_2, h, \mu)\ , \tag{3.8}$$

if in both equations, the call of the algorithm on the left side uses the same random bits as the call of the algorithm on the right side.

3.2.3. Protocols, the Adversary, and the Experiment

We now give the formal definition of a *Signature-Authenticated Two-Round Message Exchange (SA2ME) protocol* in this model. 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 is used to phrase the correctness condition (see Section 3.4.1).

- A *freshness function* is a function which, for an identity s, state information μ_s, and a time t_s, specifies a freshness interval $\varphi(s, \mu_s, t_s)$. 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) refuses to sign for the adversary.

We next describe how all these components work together. As in [BR93a], this is done by defining an appropriate notion of *experiment*, in which the protocol is running with an *adversary*. The latter is simply an arbitrary probabilistic algorithm. The experiment proceeds as follows (see Table 3.2 for details):

We assume that at the beginning of the experiment, the adversary specifies a set of identities $A \subseteq$ IDs, which has to include both the identities of oracles the adversary calls and the identities that occur in messages.

For every principal $s \in$ IDs a *server instance* Σ_s runs under the identity s. For every pair of principals $c, s \in$ IDs arbitrarily many *client instances* $\Gamma^i_{c,s}$ can run where c acts as a client and s as a server, and where i is a natural number. We let the adversary control all these instances, that is, the adversary can decide when to call such an instance, which payloads to choose, which local times are used, etc.

The experiment then works in *steps*, where in each step the adversary can perform an action (Send, Receive, Reset, Sign, Corrupt, Time), for which it provides the parameters under its control and receives the output values:

- The Send, Receive, and Reset instructions are passed to the client and server algorithms by the experiment, adding the necessary parameters and logging the output.

- The Time instruction is used to set the local clock of a principal, the only restriction is that the value of the local clock cannot be decreased by the adversary, i. e., each principal's clock is monotone.

- The Sign and Corrupt instructions are handled by a *signature oracle*, which—by abuse of notation—is denoted by the same symbol as the signature scheme, Ω. The adversary can use these two instructions to sign bit strings and corrupt a principal's key, respectively. Signing bit strings is useful, e. g., while constructing the payload for a Send instruction, see Section 2.3.2.3. But clearly, we cannot allow the adversary to use the signature oracle to sign every bit string. Therefore, the signature oracle refuses to sign bit strings belonging to the exception set specified in the protocol description.

In the experiment *traces* are recorded for each instance, which allow us to define correctness and security of a protocol, see Section 3.4. 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 server entries, the identity of the client that the server believes it is communicating with and the message handle.

Formally, the experiment $\mathsf{Exp}_{\Pi,\mathcal{A}}$ for an adversary $\mathcal{A}$ against a SA2ME protocol Π proceeds as described in Table 3.2 where we use $v \xleftarrow{R} A$ to describe assigning the output of the (randomized) algorithm A to the variable v.

1. *Select identities, generate keys, and initialize clocks.*
   Let the adversary specify a set $A \subseteq \text{IDs}$, and for each $a \in A$:
   a) Let $(pk_a^{\text{sig}}, sk_a^{\text{sig}}) \overset{R}{\longleftarrow} G()$.
   b) Send (a, pk_a^{sig}) to the adversary.
   c) Let $t_a \longleftarrow 0$.
2. *Initialize step counter and states and traces of clients and server.*
   Let $n \longleftarrow 0$.
   For each $i \in \mathbb{N}$ and $c, s \in \text{IDs}$, let $\text{tr}_{c,s}^i \longleftarrow \varepsilon$ and $\mu_{c,s}^i \longleftarrow \varepsilon$.
   For each $s \in \text{IDs}$, let $\text{tr}_s \longleftarrow \varepsilon$ and $\mu_s \longleftarrow \varepsilon$.
3. *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:
   – $\Gamma_{c,s}^i$: Send(p)
     (i) $(m, \delta, \mu) \overset{R}{\longleftarrow} \Gamma(\text{Send}, c, s, pk_{\text{IDs}}^{\text{sig}}, sk_c^{\text{sig}}, t_c, p, \mu_{c,s}^i)$,
     (ii) $\mu_{c,s}^i \longleftarrow \mu$,
     (iii) $\text{tr}_{c,s}^i \longleftarrow \text{tr}_{c,s}^i \cdot (n, \text{Send}, t_c, p, m, \delta)$,
     (iv) return (m, δ, μ) to the adversary.
   – Σ_s: Receive(m)
     (i) $(p, \delta, c, h, \mu) \overset{R}{\longleftarrow} \Sigma(\text{Receive}, s, pk_{\text{IDs}}^{\text{sig}}, sk_s^{\text{sig}}, t_s, m, \varepsilon, \mu_s)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\text{tr}_s \longleftarrow \text{tr}_s \cdot (n, \text{Receive}, t_s, p, m, \delta, c, h)$,
     (iv) return (p, δ, c, h, μ) to the adversary.
   – Σ_s: Send(p, h)
     (i) $(m, \delta, c, h', \mu) \overset{R}{\longleftarrow} \Sigma(\text{Send}, s, pk_{\text{IDs}}^{\text{sig}}, sk_s^{\text{sig}}, t_s, p, h, \mu_s)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\text{tr}_s \longleftarrow \text{tr}_s \cdot (n, \text{Send}, t_s, p, m, \delta, c, h)$,
     (iv) return (m, δ, c, h', μ) to the adversary.
   – $\Gamma_{c,s}^i$: Receive(m)
     (i) $(p, \delta, \mu) \overset{R}{\longleftarrow} \Gamma(\text{Receive}, c, s, pk_{\text{IDs}}^{\text{sig}}, sk_c^{\text{sig}}, t_c, m, \mu_{c,s}^i)$,
     (ii) $\mu_{c,s}^i \longleftarrow \mu$,
     (iii) $\text{tr}_{c,s}^i \longleftarrow \text{tr}_{c,s}^i \cdot (n, \text{Receive}, t_c, p, m, \delta)$,
     (iv) return (p, δ, μ) to the adversary.
   – Σ_s: Reset$()$
     (i) $(m, \delta, c, h, \mu) \overset{R}{\longleftarrow} \Sigma(\text{Reset}, s, pk_{\text{IDs}}^{\text{sig}}, sk_s^{\text{sig}}, t_s, \varepsilon, \varepsilon, \varepsilon)$,
     (ii) $\mu_s \longleftarrow \mu$,
     (iii) $\text{tr}_s \longleftarrow \text{tr}_s \cdot (n, \text{Reset}, t_s, \varepsilon, \varepsilon, \text{A}, \varepsilon, \varepsilon)$,
     (iv) return (m, δ, c, h, μ) to the adversary.
   – Ω: Corrupt(a)
     (i) $\text{tr}_s \longleftarrow \text{tr}_s \cdot (n, \text{Corrupt}, t_s, \varepsilon, \varepsilon, \text{A}, \varepsilon, \varepsilon)$,
     (ii) return sk_a^{sig} to the adversary.
   – Ω: Sign(a, p)
     (i) If $p \notin E^*$, return $\{p\}_{sk_a^{\text{sig}}}$, otherwise return ε to the adversary.
   – Time(a, t)
     (i) $t_a \longleftarrow \max(t_a, t)$,
     (ii) return t_a to the adversary.

Table 3.2.: Experiment $\text{Exp}_{\Pi,\mathcal{A}}$ for adversary $\mathcal{A}$ against protocol $\Pi = (\Gamma, \Sigma, \tau, \varphi, E^*)$.

3.3. The Protocol SA2ME-1

In this section, we formally define the protocol SA2ME-1 described in Section 2.5.1 within the formal framework developed in this chapter, and comment on various aspects of it.

3.3.1. Formal Definition of the Protocol

To formally define SA2ME-1, we have to specify the client and server algorithms, the time and freshness functions, and the exceptions set. We also fix some $l_{\text{nonce}} \in \mathbb{N}$ as the length of the message id's used in the protocol.

3.3.1.1. 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 $\varphi(s, (t_{\text{min}}, L), t) = \{t' \mid t_{\text{min}} + 1 \leq t' \leq t + \text{tol}_s^+\}$.

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 $t_s + \text{tol}_s^+$ 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 $m_c = \{(\mathsf{From}\colon c, \mathsf{To}\colon s', \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c^{\text{sig}}}$, with local state $\mu = (t_{\text{min}}, L)$ at local server time t_s, 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) $V(m_c, pk_c^{\text{sig}})$ returns *false*,
   c) $t \notin \varphi(s, \mu, t_s)$,
   d) $(t', r, c') \in L$ for some t', c'.
2. While $|L| \geq \text{cap}_s$,
   a) $t_{\text{min}} \longleftarrow \min\{t' \mid (t', r', c') \in L\}$,
   b) $L \longleftarrow \{(t', r', c') \in L \mid t' > t_{\text{min}}\}$.
3. $L \longleftarrow L \cup \{(t, r, c)\}$.
4. Return $(p_c, \mathsf{A}, c, r, (t_{\text{min}}, L))$.

When asked to send a payload p_s with message handle r and state information $\mu = (t_{\text{min}}, L)$, 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 set, return $(\varepsilon, \mathsf{R}, \varepsilon, \varepsilon, \mu)$.
2. $m_s \longleftarrow \{(\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_s)\}_{sk_s^{\text{sig}}}$.

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.

3.3.1.2. 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 $m_c = \{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_c)\}_{sk_c^{\text{sig}}}$ and returns (m_c, A, r). If requested to send when $\mu \neq \varepsilon$, it returns $(\varepsilon, \mathsf{R}, \mu)$.

If the algorithm receives a message $m_s = \{(\mathsf{From}\colon s', \mathsf{To}\colon c', \mathsf{Ref}\colon r', \mathsf{Body}\colon p'_s)\}_{sk_{s'}^{\text{sig}}}$ 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) $V(m_s, pk_s^{\text{sig}})$ returns *false*,
   e) $r' \neq \mu$.
2. Return $(p_s, \mathsf{A}, 0^{1+l_{\text{nonce}}})$.

3.3.1.3. Bit String Encodings and Exception Set

Our description above leaves open the actual format of the messages. We assume that our abstract messages, i. e., messages in the notation used above containing, e. g., tuples or tags like From or Time, are encoded as bit strings in such a way that the individual components can be retrieved without ambiguity. There may be multiple different bit strings that encode the same abstract message, we then call the encoded bit string equivalent.

The set $E^* \subseteq \{0,1\}^*$ is the set of all bit string encodings 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 a standard encoding for web services, SOAP [ML07,NGM+07, KMG+07], one can meet the above requirements.

This completes the formal definition of SA2ME-1. Note that it can easily be seen that the restrictions on execution orders from Section 3.2.2.2 hold. Also, it is easy to see that our protocol indeed achieves to work with bounded memory:

The size of the state of a server s in SA2ME-1 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 set 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 $c \in \{0,1\}^{l_{\text{ID}}}$.

3.3.2. Comments and Caveats

For a fixed protocol run, we 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.

3.3.2.1. Message Equivalence

Assume that $m_1 = (m'_1, \sigma_1)$ is a message created by the client or server algorithm of SA2ME-1 with a valid signature for some public key pk^{sig}, i. e. $V(m_1, pk^{\text{sig}})$ returns true. It is now easy to see that for SA2ME-1, another bit string m_2 is equivalent to m_1 as defined in Section 3.2.2.3 if and only if m_2 is of the form (m'_2, σ_2) where (i) m'_2 is a bit string encoding equivalent to m'_1 and (ii) $V(m_2, pk^{\text{sig}})$ returns true.

Therefore, messages are equivalent if they decode into equal abstract messages and if they have equal signatures or if they have different, but valid signatures from the same signature key.

3.3.2.2. Resets

From the specification of SA2ME-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 is 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 SA2ME-1 this is the case.

3.3.2.3. Parametrization

Our protocol is parameterized, since l_{nonce}, tol_s^+, and cap_s can be chosen freely, and the latter two can be chosen per server. Although the protocol is later proven correct and secure for any choice of tol_s^+ and cap_s, 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:

1. $t_c(n_1) > t_s(n_2) + \mathrm{tol}_s^+$, or
2. $t_c(n_1) \leq t'_{\min}$ where $t'_{\min}$ is the internal variable $t_{\min}$ of party s 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 capacity allows. 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.

3.3.2.4. Responding to old Messages

A protocol is only required to allow the service to respond to the most recently received and accepted message (see Section 3.2.2.2). 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 is 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 is 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 $\tau(m)$.

3.3.2.5. Dishonest Timestamps

In a way, the protocol SA2ME-1 gives the clients an 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 most likely are not 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 Section 3.6, we explain how this can be achieved.

3.4. Correctness and Security Definitions

We now define what it means that a protocol is correct and secure in our model.

Again, 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 $a \in$ 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 Ω: 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.

3.4.1. 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 delayed or 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.

A SA2ME protocol Π is (n, ε)-*correct* if for any benign adversary $\mathcal{A}$ that starts at most n many client sessions, and any $c, s \in$ IDs the following conditions are met:

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 $1 - \varepsilon$.

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 $(n', \ldots, \mathsf{A}, \ldots) \in \mathrm{tr}_s$ such that $n_2 < n' < n_3$, then $p_s = p'_s$ 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, see Section 3.6 for a discussion.

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, see Section 3.3.2.4 for some notes on this for SA2ME-1.

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.

3.4.2. Security Definition

For the security definition, we need the notion of running time of algorithms. We 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 follows: Generating a key pair takes time t_G, and signing or verifying a bit-string with l bits takes time $t_S(l)$ or $t_V(l)$, respectively.

We only need to look at 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 $(n, \ldots, \mathrm{A})$. We also say that an instance *accepts at step n* if there is an entry of the form $(n, \ldots, \mathrm{A})$ or $(n, \ldots, \mathrm{A}, \ldots)$ in its trace.

We now define when a protocol is called secure by defining a function which matches client and server traces. A *partner function* is a partial map $f\colon \mathrm{IDs} \times \mathrm{IDs} \times \mathbb{N} \to \mathbb{N}$. 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.

Depending on the result of the experiment, we then define the event $\mathrm{NoMatch}_{\Pi,\mathcal{A}}$, which is intended to model the event that the adversary $\mathcal{A}$ has "broken" the run of the experiment.

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 $(n_1, \mathrm{Send}, t_1, p_c, m_c, \mathrm{A})(n_4, \mathrm{Receive}, t_4, p_s, m_s, \mathrm{A})$ and there are timestamps t_2, t_3, step numbers $n_1 < n_2 < n_3 < n_4$, bit strings $\bar{m}_c \equiv m_c$ and $\bar{m}_s \equiv m_s$, and a handle h such that $(n_2, \mathrm{Receive}, t_2, p_c, \bar{m}_c, \mathrm{A}, c, h) \in \mathrm{tr}_s$, $(n_3, \mathrm{Send}, t_3, p_s, \bar{m}_s, \mathrm{A}, c, h) \in \mathrm{tr}_s$ as well as $f(c, s, i) = n_2$.

2. A step $(n_2, \mathrm{Receive}, t_2, p_c, m_c, \mathrm{A}, c, h)$ 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 $(n_1, \mathrm{Send}, t_1, p_c, \bar{m}_c, \mathrm{A})$ for some t_1, $n_1 < n_2$, and some $\bar{m}_c \equiv m_c$.

In Section 3.4.2.1 we explain the use of the equivalence relation in this definition.

For a partner function f, the event $\mathrm{NoMatch}^f_{\Pi,\mathcal{A}}$ (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 $(n_2, \text{Receive}, t_2, p_c, m_c, \mathsf{A}, c, h) \in \text{tr}_s$ for which no session number i exists such that the step matches the client trace $\text{tr}^i_{c,s}$ w. r. t. f.

The event $\text{NoMatch}_{\Pi,\mathcal{A}}$ denotes that the event $\text{NoMatch}^f_{\Pi,\mathcal{A}}$ occurs for *all* partial functions $f\colon \text{IDs} \times \text{IDs} \times \mathbb{N} \to \mathbb{N}$ 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

$$\text{Adv}_{\Pi,\mathcal{A}} = \Pr\left[\text{NoMatch}_{\Pi,\mathcal{A}}\right] \quad . \tag{3.9}$$

An adversary is called $(t, n_{\text{ID}}, n_{\text{rcv}}, n_{\text{send}}, n_{\text{sign}}, n_{\text{time}}, n_{\text{cor}}, l_{\text{data}})$*-adversary* if the following holds: 1. Its overall running time is bounded by t, 2. the adversary selects no more than n_{ID} identities in its first step, 3. 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, 4. the total number of principals corrupted by the adversary is not larger than n_{cor}, and 5. in each of these calls, the size of the payload or message provided to the principal is no more than l_{data}.

A SA2ME protocol Π is $(t, n_{\text{ID}}, n_{\text{rcv}}, n_{\text{send}}, n_{\text{sign}}, n_{\text{time}}, n_{\text{cor}}, l_{\text{data}}, \varepsilon)$*-secure* if we have $\text{Adv}_{\Pi,\mathcal{A}} \leq \varepsilon$ for any $(t, n_{\text{ID}}, n_{\text{rcv}}, n_{\text{send}}, n_{\text{sign}}, n_{\text{time}}, n_{\text{cor}}, l_{\text{data}})$-adversary $\mathcal{A}$.

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

Hence, our notion of security does allow a server to accept the same message twice, but only if it also has been sent twice. However, since there is no communication between server and client except for the exchanged messages, the server has no way of knowing whether a message that has been received twice was also sent twice. Therefore, protocols satisfying our security definition have to be designed in such a way that a message is accepted at most once by a server (with all but negligible probability).[10]

3.4.2.1. Message Equivalence

In the definition of matching traces, we used the equivalence relation defined in Section 3.2.2.3. This explicitly allows the adversary to modify messages on the network without breaking the security of the protocol. But as we defined above, the adversary is only allowed to replace a message with an equivalent one, i. e., it is only allowed

[10]Observe that it is of course allowed for the server to accept the same *payload* twice from the same client.

to modify a message in a way that does not modify the behavior of the protocol algorithms. Hence, the results returned to the service and the environment are the same.

For example, in the case of SA2ME-1 (see below), this security definition allows the adversary to replace the valid signature of a message with another, equivalently valid signature from the same signature key. To disallow this kind of replacement, one could drop the message equivalence relation from the definition of matching traces and force the traces of client and server to match exactly. Then, for SA2ME-1 to be asymptotically secure, one would need a signature scheme that is secure against *strong* existential unforgeability, see, e. g., [ADR02]; but we argue that this additional security guarantee does not rectify requiring a more complex signature scheme.

But message equivalence is also useful to analyze flexible encoding schemes which allow different, semantically equivalent encodings; this is, for example, heavily used in SOAP to introduce extendibility. It also allows explicit modification of messages by the network as long as it is not security relevant: For example, in SOAP, so-called intermediaries are allowed to modify some message headers while transporting a message from the sender to the final receiver.

3.5. Correctness and Security of SA2ME-1

First, we state that SA2ME-1 is indeed correct:

Theorem 3.1. *The protocol SA2ME-1 with message id length* l_{nonce} *is an* (n, ε)*-correct SA2ME protocol, where* $\varepsilon = 1 - 2^{-n \cdot l_{\text{nonce}}} \cdot \prod_{i=0}^{n+1} (2^{l_{\text{nonce}}} - i)$.

We now state that SA2ME-1 is secure. We show this by following a standard approach: For each adversary against SA2ME-1 we construct an adversary against the underlying signature scheme with comparable running time and success probability. Recall the notion of a signature scheme that is (t, q, l, ε)-secure against EUF-CMA from Section 2.1.3.1.

Theorem 3.2. *SA2ME-1 is* $(t_1, n_{\text{ID}}, n_{\text{rcv}}, n_{\text{send}}, n_{\text{sign}}, n_{\text{time}}, n_{\text{cor}}, l_{\text{data}}, \varepsilon_1)$-secure *if the signature scheme used is* $(t_2, q_2, l_2, \varepsilon_2)$*-secure with*

$$t_2 \in O(t_1 + n_{\text{ID}} \cdot (t_G + n_{\text{ops}} \cdot (\text{cap}_{\max} \cdot (l_{\text{ID}} + l_{\text{time}}) + t_S(l_{\text{msg}})))), \tag{3.10}$$

$$q_2 \leq n_{\text{sign}} + n_{\text{send}}, \tag{3.11}$$

$$l_2 \leq l_{\text{msg}}, \tag{3.12}$$

$$\varepsilon_2 \leq \frac{\varepsilon_1}{n_{\text{ID}}} + \frac{2^{l_{\text{nonce}}}!}{(2^{l_{\text{nonce}}} - n_{\text{ID}} \cdot n_{\text{send}})! \cdot 2^{l_{\text{nonce}} \cdot n_{\text{ID}} \cdot n_{\text{send}}}} - 1 \ , \tag{3.13}$$

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

The security proof for our protocol, see Section 3.5.2, first establishes that in SA2ME-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 $\text{NoMatch}_{\Pi,\mathcal{A}}$) 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 SA2ME-1 and a "simulated" protocol environment to construct an adversary against the signature scheme. Theorem 3.2 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 3.2 are reasonably small.

Since Cook and Reckhow proved that RAM machines and Turing machines are polynomially equivalent [CR73], the above Theorem 3.2 implies the following:

Corollary 3.3. *SA2ME-1 is asymptotically secure if it uses a signature scheme that is asymptotically EUF-CMA secure.*

3.5.1. Correctness of SA2ME-1

Proof of Theorem 3.1. As a first step, we note that there are $2^{l_{\text{nonce}}}$ different message id's, hence the probability of n message id's chosen uniformly at random being different is exactly

$$\frac{2^{l_{\text{nonce}}} \cdot (2^{l_{\text{nonce}}} - 1) \cdot (2^{l_{\text{nonce}}} - 2) \cdot \dots \cdot (2^{l_{\text{nonce}}} - n + 1)}{(2^{l_{\text{nonce}}})^n}, \tag{3.14}$$

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 $(n_1, \text{Send}, t_1, p_c, m_c, \text{A}) \in \text{tr}^i_{c,s}$ and $(n_2, \text{Receive}, t_2, p'_c, m_c, \delta_s, c', h) \in \text{tr}_s$ 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 $(n_3, \text{Send}, t_3, p_s, m_s, \text{A}, c', h) \in \text{tr}_s$ and $(n_4, \text{Receive}, t_4, p'_s, m_s, \delta_c) \in \text{tr}^i_{c,s}$ with $n_2 < n_3$ and $n_1 < n_4$, but with no $(n', \dots, \text{A}, \dots) \in \text{tr}_s$ such that $n_2 < n' < n_3$.

First, we know that the server only generates one response for the incoming message m_c (as it 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. □

3.5.2. Security of SA2ME-1

To prove Theorem 3.2, we perform a concrete security analysis of SA2ME-1: We show that an adversary with a given resource bound and success probability against SA2ME-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 SA2ME-1.

We proceed in two steps: We first show that every successful attack against SA2ME-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 SA2ME-1 is secure in an asymptotic sense.

In a second step, for a more detailed analysis, we provide a simulator $\mathcal{S}$ (see Appendix A and the explanation in Section 3.5.2.2) which turns any adversary $\mathcal{A}$ against SA2ME-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.

3.5.2.1. Attack Implies Collision or Forgery

Lemma 3.4. *Let $\mathcal{A}$ be an adversary. For every run of the experiment $\mathsf{Exp}_{\text{SA2ME-1},\mathcal{A}}$ in which the event $\mathsf{NoMatch}_{\text{SA2ME-1},\mathcal{A}}$ occurs, one of the following events occurs as well:*

(a) $\mathcal{A}$ produced a bit string that is accepted as a valid signature for an uncorrupted identity, which was neither obtained from the client or server algorithms nor from the signature oracle, or

(b) 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.

Note that to achieve the properties mentioned in the lemma, 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 Lemma 3.4, we first show that SA2ME-1

is resistant against replay attacks. The following lemma states that equivalent messages are not accepted twice by a server during a protocol run:

Lemma 3.5. *Let $\mathcal{A}$ be an adversary and $s \in \text{IDs}$. Then in every run of $\mathsf{Exp}_{\text{SA2ME-1},\mathcal{A}}$, 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 $m_1 \equiv m_2$, then $n_1 = n_2$ and thus $m_1 = m_2$.*

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.

Proof of Lemma 3.5. Assume that a server s accepts equivalent messages $m_1 \equiv m_2$ that both decode into $\{(\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon x)\}_{sk_c^{\text{sig}}}$, 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 m_2, 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_1 with timestamp t at point n_1, we know that $t \leq t_s(n_1) + \text{tol}^+_s$. We also know that $t_s(n_r) + \text{tol}^+_s \leq t^s_{\min}(n_2)$, since the server runs SA2ME-1. Therefore we conclude $t^s_{\min}(n_2) < t \leq t_s(n_1) + \text{tol}^+_s \leq t_s(n_r) + \text{tol}^+_s \leq t^s_{\min}(n_2)$—a contradiction. □

We remark 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.

Proof of Lemma 3.4. Let $\mathcal{A}$ be an adversary. Fix a run of the experiment $\mathsf{Exp}_{\text{SA2ME-1},\mathcal{A}}$ in which the event $\mathsf{NoMatch}_{\text{SA2ME-1},\mathcal{A}}$ appears.

By construction of the experiment, every signature for a valid SA2ME-1 message that $\mathcal{A}$ 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 Ω refuses to sign these bit strings.

We now define a partner function f as follows: For every client instance $\Gamma^i_{c,s}$, if the first accepting step in $\text{tr}^i_{c,s}$ (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 Receive query of the server instance Σ_s with some incoming message $\bar{m} \equiv m$, if such a step exists; let $f(c, s, i)$ be undefined otherwise.

By the prerequisites, we know that $\mathsf{NoMatch}^f_{\text{SA2ME-1},\mathcal{A}}$ occurs in the protocol run. We assume that neither (a) existential forgery against an uncorrupted key nor (b) collision of message id's for client sessions $\Gamma^i_{c,s}$ and $\Gamma^{i'}_{c,s}$ for $i \neq i'$ occurs.

Note that under the above assumptions, if equivalent messages $m_1 \equiv m_2$ occur in the protocol run, we know that the signatures on both m_1 and m_2 are the same, i. e., if the messages differ, then only on the encoding of non-signed parts.

To prove this, let messages $m_1 \equiv m_2$ be messages occurring in the run of the protocol. Then we know that the abstract messages in the notation used above is the same for both messages. Now if m_1 and m_2 are request messages, they both were signed by the same client instance, as by assumption no collision of message id's and no forgery occurred; hence, the signatures of both messages are the same. If, on the other hand, m_1 and m_2 are response messages, they also contain the same signature, as each server sends at most one response containing a certain message id (again, by assumption no collision of message id's occurred).

Thus, for the rest of this proof, without loss of generality, we assume that if messages signed by uncorrupted keys are equivalent, they are also equal.

We prove the lemma by distinguishing the two cases in the definition of $\mathsf{NoMatch}^f$ (see Section 3.4.2) and leading both cases to a contradiction.

First Case Assume that case (a) in the definition of $\mathsf{NoMatch}^f_{\mathsf{SA2ME\text{-}1},\mathcal{A}}$ occurs. By definition of the NoMatch 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 hat been sent by a server with s's signature. Note that SA2ME-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 Receive request accepted by s with a matching message handle (i. e., a message id). Hence there is an entry $(n_2, \mathsf{Receive}, t_2, p'_c, m'_c, \mathsf{A}, c'', h)$ in the trace tr_s with $n_2 < n_3$, and there is no accepted Receive instruction or Send instruction with message handle h in tr_s with a step number between n_2 and n_3. By construction of the protocol, it follows

that $c'' = c$. Since Σ_s accepts the message m'_c and determines the sender to be $c'' = c$, it follows that m'_c is a valid SA2ME-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, in the client trace $\mathrm{tr}^{i'}_{c,s}$, there is an entry $(n'_1, \mathsf{Send}, t'_1, p''_c, m'_c, \mathsf{A})$ with $n'_1 < n_2$. 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 SA2ME-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 3.5, 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 SA2ME-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 3.5, 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. □

3.5.2.2. Concrete Analysis

Proof of Theorem 3.2. Let Π denote the protocol SA2ME-1. As noted above, we provide a simulator $\mathcal{S}$ (see Appendix A) which turns an adversary $\mathcal{A}$ against SA2ME-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 $(t, n_{\text{ID}}, n_{\text{rcv}}, n_{\text{send}}, n_{\text{sign}}, n_{\text{time}}, n_{\text{cor}}, l_{\text{data}})$-adversary which has an advantage $\text{Adv}_{\Pi,\mathcal{A}}$ against the protocol SA2ME-1. Let $\Omega = (G, S, V)$ be the signature scheme used in the protocol. We analyze the adversary $\mathcal{S}_{\mathcal{A}}$ against the signature scheme Ω. Thus, $\mathcal{S}_{\mathcal{A}}$ is given a public key $pk_\star^{\text{sig}}$ and access to a signature oracle $\Omega_\star$; to successfully break the signature scheme, it has to provide a message m and a signature σ such that $V((m, \sigma), pk_\star^{\text{sig}})$ returns true.

We sketch what the simulator $\mathcal{S}_{\mathcal{A}}$ in Appendix A does. Roughly speaking, it runs the experiment from Table 3.2, where it replaces one (randomly chosen) public key with $pk_\star^{\text{sig}}$.

More precisely, the simulator starts by letting the adversary choose a set of identities A (a subset of the set IDs as in the experiment). Then, the simulator randomly selects one of these identities, which we call x, and generates a signature key pair for all identities in $A \setminus \{x\}$. Now the simulator is able to sign messages and verify signatures for all identities $a \in A$: If $a \neq x$, the simulator uses the generated key pair, but if $a = x$, the simulator uses the signature oracle $\Omega_\star$ to sign messages and the public key $pk_\star^{\text{sig}}$ to verify messages.

Now, $\mathcal{S}_{\mathcal{A}}$ simulates the adversary $\mathcal{A}$. If, during this simulation, $\mathcal{A}$ requests a client or a server to perform a step like Send, Receive, Reset, or Time, the simulator runs the algorithms of the protocol as specified in the experiment in Table 3.2, using the adversary's parameters as input and handing the algorithm's output back to the adversary as defined in the experiment, and generating and verifying signatures as described above. If $\mathcal{A}$ performs a Sign query to its signature oracle Ω, the simulator either uses one of the generated key pairs or relays the query to the signature oracle $\Omega_\star$. If $\mathcal{A}$ chooses to corrupt the private key of some identity $a \in A$, the simulator hands the corresponding signature key to the adversary if $a \neq x$, and stops otherwise (in which case $\mathcal{S}_{\mathcal{A}}$ fails to break the signature scheme).

In any case, the simulator logs all calls to the oracle $\Omega_\star$; and each time a signature is verified using key $pk_\star^{\text{sig}}$, the simulator checks if the verification is successful *and* the message has not been logged before: If both conditions hold, the simulator found a forgery. Thus, if the adversary $\mathcal{A}$ is successful in breaking the protocol because it manages to forge a valid signature which has not been produced by $\Omega_\star$ and thus not logged, the simulator can detect this forgery and output it; if not, the simulator fails.

Success probability We analyze the advantage $\text{Adv}_{\Omega,\mathcal{S}_{\mathcal{A}}}$ of the simulator $\mathcal{S}_{\mathcal{A}}$ against the signature scheme Ω.

We define $\text{NoMatch}_{\Pi,\mathcal{A}}^{a}$ to be the event where the adversary is successful against an identity a, either by forging a signature under a's identity without corrupting a's private

key, or because two client instances of a chose colliding message id's. Due to Lemma 3.4 we know that $\mathsf{NoMatch}_{\Pi,\mathcal{A}} = \bigcup_{a\in A} \mathsf{NoMatch}^a_{\Pi,\mathcal{A}}$. Thus, we have

$$\mathrm{Adv}_{\Pi,\mathcal{A}} = \Pr(\mathsf{NoMatch}_{\Pi,\mathcal{A}}) \leq \sum_{a\in A} \Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}}) \ . \tag{3.15}$$

Now let $\mathcal{S}^a_{\mathcal{A}}$ be the variant of the simulator $\mathcal{S}_{\mathcal{A}}$ that replaces a's public key with $pk^{\mathrm{sig}}_{\star}$. This simulator is successful against the signature scheme Ω if the event $\mathsf{NoMatch}^a_{\Pi,\mathcal{A}}$ occurs, but no message id's collide, which we denote by $\overline{\mathsf{Coll}_{\Pi,\mathcal{A}}}$. Thus, we have

$$\mathrm{Adv}_{\Omega,\mathcal{S}^a_{\mathcal{A}}} \geq \Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}} \cap \overline{\mathsf{Coll}_{\Pi,\mathcal{A}}}) \ . \tag{3.16}$$

The probability $\Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}} \cap \overline{\mathsf{Coll}_{\Pi,\mathcal{A}}})$ is at least $\Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}}) - \Pr(\mathsf{Coll}_{\Pi,\mathcal{A}})$, where $\mathsf{Coll}_{\Pi,\mathcal{A}}$ denotes that a collision occurred.

As the simulator $\mathcal{S}_{\mathcal{A}}$ chooses some $a \in A$ at random and replaces a's public key with $pk^{\mathrm{sig}}_{\star}$, we have

$$\begin{aligned}
\mathrm{Adv}_{\Omega,\mathcal{S}_{\mathcal{A}}} &= \frac{1}{n_{\mathrm{ID}}} \sum_{a\in A} \mathrm{Adv}_{\Omega,\mathcal{S}^a_{\mathcal{A}}} && (3.17)\\
&\geq \frac{1}{n_{\mathrm{ID}}} \sum_{a\in A} \Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}} \cap \overline{\mathsf{Coll}_{\Pi,\mathcal{A}}}) && (3.18)\\
&\geq \frac{1}{n_{\mathrm{ID}}} \sum_{a\in A} \left(\Pr(\mathsf{NoMatch}^a_{\Pi,\mathcal{A}}) - \Pr(\mathsf{Coll}_{\Pi,\mathcal{A}})\right) && (3.19)\\
&\geq \frac{1}{n_{\mathrm{ID}}} \Pr(\mathsf{NoMatch}_{\Pi,\mathcal{A}}) - \Pr(\mathsf{Coll}_{\Pi,\mathcal{A}}) && (3.20)\\
&= \frac{\mathrm{Adv}_{\Pi,\mathcal{A}}}{n_{\mathrm{ID}}} - \Pr(\mathsf{Coll}_{\Pi,\mathcal{A}}) \ . && (3.21)
\end{aligned}$$

Finally, the probability $\Pr(\mathsf{Coll}_{\Pi,\mathcal{A}})$ can be calculated as follows: For each Send action of a client, one message id of length l_{nonce} is randomly chosen. Thus, at most $n_{\mathrm{ID}} \cdot n_{\mathrm{send}}$ message id's are chosen from a set of size $2^{l_{\mathrm{nonce}}}$. The resulting probability of a collision is given by

$$\Pr(\mathsf{Coll}_{\Pi,\mathcal{A}}) = 1 - \frac{2^{l_{\mathrm{nonce}}}!}{(2^{l_{\mathrm{nonce}}} - n_{\mathrm{ID}} \cdot n_{\mathrm{send}})! \cdot 2^{l_{\mathrm{nonce}} \cdot n_{\mathrm{ID}} \cdot n_{\mathrm{send}}}} \ . \tag{3.22}$$

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 $t_S(l) \in \Omega(l)$.

$$
\begin{aligned}
t_{\mathtt{time}} &\in O(t_{\mathtt{userNr}}) \\
t_{\mathtt{clientSend}} &\in O(t_{\mathtt{userNr}} + t_{\mathtt{sign}}) \\
t_{\mathtt{serverReceive}} &\in O(t_{\mathtt{userNr}} + t_{\mathtt{verify}} + \mathrm{cap}_{\mathrm{max}} \cdot (l_{\mathrm{ID}} + l_{\mathrm{time}}) \\
&\qquad + t_{\mathtt{map}}(\mathrm{cap}_{\mathrm{max}}, l_{\mathrm{nonce}})) \\
t_{\mathtt{serverSend}} &\in O(t_{\mathtt{userNr}} + t_{\mathtt{sign}} + t_{\mathtt{map}}(\mathrm{cap}_{\mathrm{max}}, l_{\mathrm{nonce}})) \\
t_{\mathtt{clientReceive}} &\in O(t_{\mathtt{userNr}} + t_{\mathtt{verify}}) \\
t_{\mathtt{corrupt}} &\in O(t_{\mathtt{userNr}}) \\
t_{\mathtt{sign}} &\in O(t_{\mathtt{userNr}} + t_S(l_{\mathrm{msg}}) + t_{\mathtt{map}}(n_{\mathrm{sign}} + n_{\mathrm{send}}, l_{\mathrm{msg}})) \\
t_{\mathtt{verify}} &\in O(t_{\mathtt{userNr}} + t_V(l_{\mathrm{msg}}) + t_{\mathtt{map}}(n_{\mathrm{sign}} + n_{\mathrm{send}}, l_{\mathrm{msg}})) \\
t_{\mathtt{userNr}} &\in O(t_{\mathtt{map}}(n_{\mathrm{ID}}, l_{\mathrm{ID}}))
\end{aligned}
$$

Table 3.3.: Running times of the procedures of the simulator

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 $t_{\mathtt{map}}(n, l)$ 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 $(\mathrm{o}, n) \in \{(\mathrm{tuple}, 2), (\mathrm{request}, 5), (\mathrm{response}, 4), (\mathrm{signature}, 2)\}$ and all bit strings $\beta_1, \ldots, \beta_n$, we assume that $D(\mathrm{o}, E(\mathrm{o}, \beta_1, \ldots, \beta_n)) = (\beta_1, \ldots, \beta_n)$ as well as $|E(\mathrm{o}, \beta_1, \ldots, \beta_n)| \in O(\sum_{i=1}^{n} |\beta_i|)$.

The running time of the single functions can be bounded as shown in Table 3.3 for a fixed $l_{\mathrm{msg}} \in O(l_{\mathrm{ID}} + l_{\mathrm{nonce}} + l_{\mathrm{time}} + l_{\mathrm{data}})$. Then the overall running time of the simulator $\mathcal{S}_{\mathcal{A}}$, denoted t, is as follows, where $n_{\mathrm{ops}} = n_{\mathrm{rcv}} + n_{\mathrm{send}} + n_{\mathrm{sign}} + n_{\mathrm{time}}$:

$$
\begin{aligned}
&O(t_{\mathcal{A}} + n_{\mathrm{ID}} \cdot t_G + n_{\mathrm{cor}} \cdot t_{\mathtt{corrupt}} + n_{\mathrm{ID}} \cdot n_{\mathrm{sign}} \cdot t_{\mathtt{sign}} + n_{\mathrm{ID}} \cdot n_{\mathrm{time}} \cdot t_{\mathtt{time}} && (3.23)\\
&\quad + n_{\mathrm{ID}} \cdot (n_{\mathrm{send}} \cdot (t_{\mathtt{clientSend}} + t_{\mathtt{serverSend}}) + n_{\mathrm{rcv}} \cdot (t_{\mathtt{clientReceive}} + t_{\mathtt{serverReceive}})) \\
=&O(t_{\mathcal{A}} + n_{\mathrm{ID}} \cdot (t_G + n_{\mathrm{ops}} \cdot (t_S(l_{\mathrm{msg}}) + t_{\mathtt{map}}(\mathrm{cap}_{\mathrm{max}}, l_{\mathrm{nonce}}) && (3.24)\\
&\quad + t_{\mathtt{map}}(n_{\mathrm{sign}} + n_{\mathrm{send}}, l_{\mathrm{msg}}) + t_{\mathtt{map}}(n_{\mathrm{ID}}, l_{\mathrm{ID}})) + \mathrm{cap}_{\mathrm{max}} \cdot (l_{\mathrm{ID}} + l_{\mathrm{time}}))) \\
=&O(t_{\mathcal{A}} + n_{\mathrm{ID}}(t_G + n_{\mathrm{ops}}(t_S(l_{\mathrm{msg}}) + \mathrm{cap}_{\mathrm{max}}(l_{\mathrm{ID}} + l_{\mathrm{time}})))). && (3.25)
\end{aligned}
$$

Note that the machine model we use would allow us to address arbitrary registers, e. g., we could directly use bit strings (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. However, our simulator only uses these capabilities of the model in the standard way. In particular, the adversary $\mathcal{S}_\mathcal{A}$ obtained by our construction is a natural and realistic adversary.

Finally, note that the simulator $\mathcal{S}_\mathcal{A}$ makes at most $n_{\text{sign}} + n_{\text{send}}$ 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 $(n_{\text{sign}} + n_{\text{send}}) \cdot l_{\text{msg}}$. □

This concludes the security proof for SA2ME-1 in this model.

3.6. Practical Choice of Parameters

A weakness of the protocol as stated and discussed in Section 3.3.2 are the rather vague guarantees implied by our security definition: 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 these. In the following lemma, we specify one way of choosing values for these parameters such that "liveliness" of the servers is guaranteed at all times.

Lemma 3.6. *Let s be a SA2ME-1 server, let tol_s^+ be the server's tolerance, and let t_{diff} be at least the time span (measured by the server's local clock) 1. between accepting two messages as well as 2. between a reset and accepting the first message. Then, if tol_s^- is a real number such that*

$$\text{cap}_s > \frac{\text{tol}_s^+ + \text{tol}_s^-}{t_{\text{diff}}} \quad , \tag{3.26}$$

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

The lemma establishes 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 lemma 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 3.3.2, 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.

Proof of Lemma 3.6. 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: If $t \geq t_r^s + \text{tol}_s^+ + \text{tol}_s^-$, 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. □

4. Simulation-Based Analysis

The second framework that we use to analyze all three protocols is the *Inexhaustible Interactive Turing Machines (IITM)* framework introduced by Küsters in [Kü06a]. In this chapter, we uniformly model the security guarantees fulfilled by our three protocols in an ideal functionality and we give implementations of the protocols that are later proven secure.

In Section 4.1, we briefly introduce the IITM framework. We then define a ideal functionality for S2ME protocols in Section 4.2 that is (through parameterization) flexible enough to capture the functionality of all three protocols. Next, in Section 4.3, we provide three different implementations of different parameterizations of the ideal functionality; in Section 4.4 these three implementations are shown to securely realize the ideal functionality. As our implementations use idealized versions of cryptographic primitives, we make some remarks on realizing those in Section 4.5. We conclude the chapter with some comments in Section 4.6.

Parts of the results in this chapter, namely an analysis of a simpler modeling of SA2ME-1, were published in [KSW09b, KSW09c], see Section 4.6.5 for some remarks on differences to the work in this chapter.

4.1. Simulation-Based Security and the IITM Framework

Simulation-based security allows to analyze cryptographic protocols such that properties proven remain true even when the protocol is used as a sub-protocol of a larger system.

The main idea is to define a so-called *ideal functionality*, which specifies a cryptographic goal to be realized by a protocol in an idealized fashion. This ideal functionality also documents the capabilities of an attacker on the protocol. A concrete protocol is called secure if it *realizes* the ideal functionality such that every attacker on the real protocol can be *simulated* in the ideal setting.

We briefly sketch Küsters' Inexhaustible Interactive Turing Machines framework. For precise definitions and background on these notions, see [Kü06a,Kü06b]. For references to similar frameworks, see the section on related work in Chapter 1.

4.1.1. Inexhaustible Interactive Turing Machines

In the IITM framework, cryptographic protocols are modeled as a set of concurrently running machines, called a *system of IITM's* (see below). The machines in the system are activated sequentially, where at each point in time, only a single machine is active,

and each machine may be activated repeatedly. A single IITM is a probabilistic Turing machine with an associated polynomial q used to bound its running time and output length.

Tapes. In addition to work tapes, an IITM has named external tapes which may be shared with other machines. External read-tapes of machines are partitioned into *consuming* and *enriching* tapes. This distinction serves to allow the maximal running time of the machines to depend on the input on the enriching tapes, and not merely on the security parameter alone as in standard cryptographic models as [BR93a].

In order to avoid an exponential blow-up of lengths of exchanged messages, a *well-formed* system is defined to be one where the sub-graph of machines connected with enriching tapes is acyclic. As proven in [Kü06a], a well-formed system can be simulated on a single polynomial-time machine.

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

Modes of Computation. An IITM can run in two different modes (determined by the content of the mode tape upon activation):

The CheckAddress mode is used to determine whether an incoming message is intended for the current machine. When activated in this mode, the IITM reads an input message from a special input tape and returns accept or reject on a special output tape.

In this mode, computation must not be probabilistic, and the number of steps taken must be bounded by $q(n)$, where q is the polynomial associated with the machine, and n is the length of the content of the work tapes, the current input, and the security parameter. This mode is typically used to verify whether an incoming message belongs to the correct session.

The Compute mode is then used for the actual computation (which may include responding to the incoming message). The number of steps in this mode must be bounded by $q(n)$, where q and n are as above.

Additionally, the total output up to a point in the run of the machine, as well as the length of all work tapes must always be bounded by $q(m)$, where m is the sum of the security parameter plus the length of all input received on enriching input tapes in mode Compute in the current run of the system.

This implies that when a machine is required to produce "long" output, it previously must be given the corresponding resources via enriching input tapes.

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

4.1.2. Systems of IITM's for Cryptographic Protocols

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

$$\mathcal{M} = M_1 \mid \ldots \mid M_k \mid !M'_1 \mid \ldots \mid !M'_k \ , \tag{4.1}$$

where the M_i and M'_i are IITM's. The machines $M'_1, \ldots, M'_k$ are said to appear in the *scope of a bang*: The bang operator "!" provides an "infinite supply" of machines (running the code of) M'_i. In a run of a system, this is handled as follows:

When a machine M sends a message (via a shared tape) to a machine M' of which one or more copies are already running, but all running copies reject the message in its CheckAddress mode and M' appears in the scope of a bang, then a new instance of M' is started, which then may accept the message in CheckAddress mode. If it does, it remains active and processes the incoming message. Otherwise it is deactivated again. This allows to start an unbounded number of sessions of a protocol.

Composition of Machines. An *external* tape of a system $\mathcal{M}$ is a tape which is a network- or I/O-tape of one of its machines for which there is no corresponding output or input tape in the system itself. These tapes allow external machines to communicate with $\mathcal{M}$, and thus enable $\mathcal{M}$ to provide a functionality to "outside" machines.

This mechanism allows to naturally compose systems of IITM's in a way allowing interaction: For two systems $\mathcal{M}_1$ and $\mathcal{M}_2$, the composition $\mathcal{M}_1 \mid \mathcal{M}_2$ denotes the system containing all machines of $\mathcal{M}_1$ and $\mathcal{M}_2$, where non-external tapes of the systems are consistently renamed (the systems only influence each other via their communication on their external tapes).

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

Equivalence By $\mathcal{M}(1^\eta, a)$ we denote running the system $\mathcal{M}$ with security parameter η and *auxiliary input* a. A system may have a special external output tape named decision. When a machine writes output to this tape (the output must be either 0 or 1), the run of the system stops immediately. With $\Pr(\mathcal{M}(1^\eta, a) \rightsquigarrow 1)$ we denote the probability that a run of $\mathcal{M}(1^\eta, a)$ results in 1 being the value written on the decision tape.

Two systems $\mathcal{M}_1$ and $\mathcal{M}_2$ are *computationally indistinguishable* if

$$|\Pr(\mathcal{M}_1(1^\eta, a) \rightsquigarrow 1) - \Pr(\mathcal{M}_2(1^\eta, a) \rightsquigarrow 1)| \tag{4.2}$$

is negligible in the security parameter η for all $a \in \{0,1\}^*$.

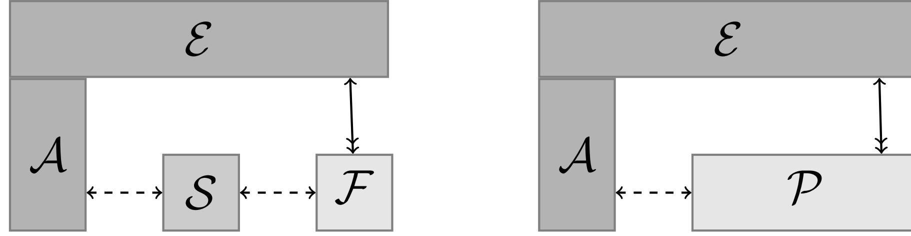

Figure 4.1.: An abstract view of the two systems of IITM's

4.1.3. Protocol Security in the IITM Framework

To define security notions for cryptographic protocols, the composition of a given system with an *environment* and an *adversary* is studied.

An *adversary* for $\mathcal{M}$ is a system $\mathcal{A}$ such that the set of external I/O-tapes of $\mathcal{M}$ and $\mathcal{A}$ are disjoint, and for every external network output tape of $\mathcal{M}$, there is an external network input tape of $\mathcal{A}$, and vice versa. This means that an adversary for $\mathcal{M}$ is syntactically suited to connect to all external "network ports" of $\mathcal{M}$. Typically, all incoming external tapes of an adversary are defined to be enriching.

An *environmental* system for $\mathcal{M}$ similarly connects to the I/O-tapes, and its set of external network tapes is disjoint with that of $\mathcal{M}$. When $\mathcal{P}$ and $\mathcal{F}$ are systems (the real and the ideal system), then an adversarially connectable system $\mathcal{S}$ is a *simulator for $\mathcal{F}$ and $\mathcal{P}$*, if $\mathcal{S} \mid \mathcal{F}$ has the exact same set of external tapes (with matching type and direction) as $\mathcal{P}$.

Note that an output (input) tape in $\mathcal{S} \mid \mathcal{F}$ is only external when there is no input (output) tape with the same name in $\mathcal{S}$ *or* in $\mathcal{F}$. Hence a simulator only connects to the network tapes of $\mathcal{F}$, and syntactically, $\mathcal{S} \mid \mathcal{F}$ and $\mathcal{P}$ "look the same". In particular, a system $\mathcal{E}$ is a suitable environment for $\mathcal{P}$ if and only if it is one for $\mathcal{S} \mid \mathcal{F}$.

We now define the central security notion that we study, also see Figure 4.1. In the following, $\mathcal{F}$ is supposed to be an *ideal* system (also called *ideal functionality*), and $\mathcal{P}$ a concrete system that attempts to *realize* the ideal functionality. $\mathcal{P}$ and $\mathcal{F}$ are I/O-compatible if they have disjoint sets of external network tapes, the same set of external I/O-tapes, and each external I/O-tape has the same direction in both.

Let $\mathcal{P}$ and $\mathcal{F}$ be I/O-compatible systems. Then $\mathcal{P}$ *securely realizes*[11] $\mathcal{F}$, denoted by $\mathcal{P} \leq^{\mathrm{BB}} \mathcal{F}$, if there is a simulator $\mathcal{S}$ for $\mathcal{P}$ and $\mathcal{F}$ such that for all adversaries $\mathcal{A}$ and environments $\mathcal{E}$ for $\mathcal{P}$ or $\mathcal{S} \mid \mathcal{F}$, the systems $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}$ and $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{F}$ are computationally indistinguishable.

This models the intuition expressed above: The simulator $\mathcal{S}$ essentially makes the system $\mathcal{F}$ behave exactly as $\mathcal{P}$ (without the simulator). Hence any attack that can be mounted on the real protocol system $\mathcal{P}$ is also successful against the ideal functionality $\mathcal{F}$.

[11] We use black-box simulatability (hence, the "BB"), see [Kü06b] for other variants and their relation.

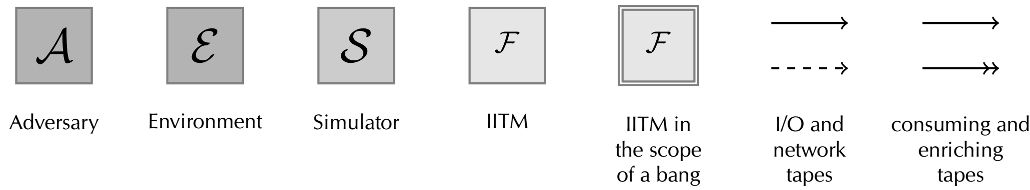

Figure 4.2.: Legend for illustrations of (systems of) IITM's

4.1.4. Notation for IITM's

We introduce some conventions we use when we define an IITM M below by listing parameters, tapes, variables, steps, functions, and CheckAddress conditions. See Figure 4.2 for conventions when illustrating (systems of) IITM's.

Parameters. Some IITM's are parameterized, e. g., we write $M(x, y)$ and view x and y as parameters influencing M's operation. If M is parameterized, we first define the names and types of the parameters.

Tapes. We list all tapes of M. We denote by $A \longleftrightarrow B$ a tape or a pair of tapes in the following way:

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

We often use the convention that the tapes connecting machines $\mathcal{X}$ and $\mathcal{Y}$ are labeled $X_a \longleftrightarrow Y_b$, $X_c \longleftrightarrow Y_d$, and so on. When we say "received from X_a"/"send to Y_b", we mean "received on the incoming tape labeled X_a" or "sent on the outgoing tape labeled Y_b".

Variables and Initialization. We list all variables that are *global* to M in the sense they are stored on the work tapes between steps (see below), and we give the values they are initialized with upon first activation; $\perp$ is used to denote an uninitialized variable or "null" value. Variables that are not listed here, but later used in the steps, are supposed to be local to that step.

Steps. The functionality of M is described as a loop containing a number of steps, where each step s has (roughly) the form

$$\texttt{if } p_s \text{ received from } a_s \text{ [while } c_s\text{], } \texttt{do } L_s \tag{4.3}$$

where p_s is some pattern of a message, a_s is an identity, c_s may contain some additional conditions, and L_s is a list of instructions.

We assume that for each incoming message m, the machine M looks for the first step s that has a matching pattern p_s, where the sender of m matches a_s and where the additional conditions c_s hold (similar to the concept of guarded commands, see [Dij75]). In the pattern, we underline variables to denote binding on first match, i. e., an underlined variable $\underline{x}$ denotes "match any value v and then let $x = v$", whereas a simple x means "only match messages that contain the value of x".

If no matching step is found, m is ignored; otherwise, if step s matches, the instructions in L_s are executed. If L_s contains instructions to receive a message matching some pattern p', the machine waits for exactly that message: M accepts any message m' that the CheckAddress mode permits (see below), but M then drops that message m' if it does not match p' (even if m' matches the pattern $p_{s'}$ of some steps s'). This ensures that the processing of one step can only be interrupted, but not canceled.

In addition to global variables as defined above, the instructions in L_s may contain *local* variables in the sense that they are only valid throughout L_s and discarded after the execution of L_s.

If a step contains the instruction "break", the machine stops the current execution of the current step, i. e., the machine terminates, but it accepts new messages after that. In contrast, the instruction "halt" terminates a machine completely, i. e., from that point on, the CheckAddress mode rejects all messages.

Functions. For some functionalities we also define functions or subroutines. In addition to the usual calls to functions or subroutines, we sometimes run subroutines "concurrently":

We assume that the processing of the subroutine starts immediately, but the subroutine may include instructions to receive certain messages. If that is the case, the subroutine (and the simulator) pauses execution. Now, if some message is received by the simulator, it checks if one of the subroutines is waiting for that message, and if that is the case, activates the corresponding subroutine. Otherwise, the message is processed as usually by the steps of the simulator functionality, see above.

Thus, the subroutines and the normal steps of the simulator are not really executed concurrently, but in an interleaving semantics, just as IITM's.

CheckAddress. Finally, for most functionalities we describe the IITM's operation in CheckAddress mode. If this part is left out, the machine accepts all incoming messages.

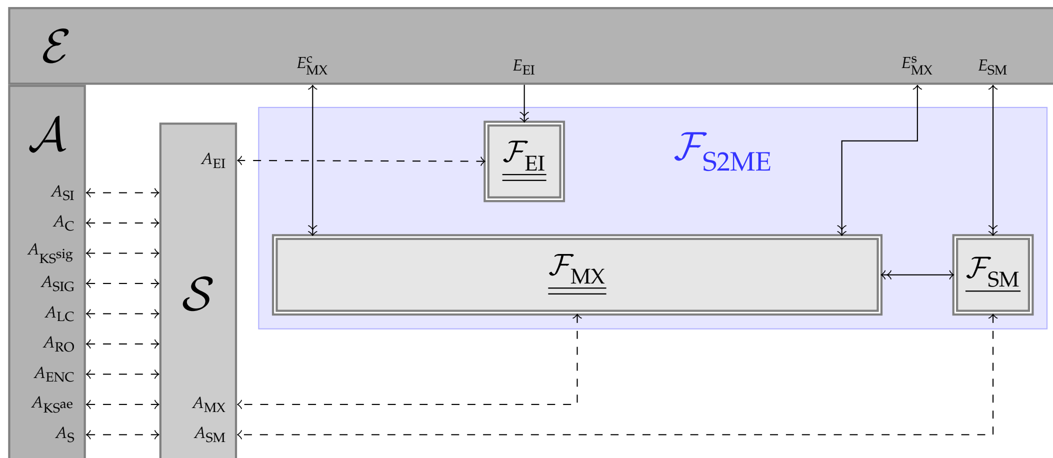

Figure 4.3.: The ideal functionality $\mathcal{F}_{S2ME}$ connected to the environment and the simulator, which is in turn connected to the adversary.

4.2. Ideal Functionality for Secure Two-Round Message Exchange

In this section, we introduce an ideal functionality $\mathcal{F}_{S2ME}$ for secure two-round message exchange. It consists of the *Message Exchange* functionality ($\mathcal{F}_{MX}$), the *Server Management* functionality ($\mathcal{F}_{SM}$), and the *Enriching Input* functionality ($\mathcal{F}_{EI}$), given in Appendices B.1.1, B.1.2, and B.1.3, respectively. The ideal functionality is parameterized as described in Section 4.2.1. For an illustration of the ideal functionality and its connection to the environments as well as the simulator that is later used in the proofs in this chapter, see Figure 4.3.

Formally, let

$$\mathcal{F}_{S2ME}(leak, pw\text{-}auth) = !\underline{\underline{\mathcal{F}_{MX}(leak, pw\text{-}auth)}} \mid !\underline{\mathcal{F}_{SM}(pw\text{-}auth)} \mid !\underline{\underline{\mathcal{F}_{EI}}}. \tag{4.4}$$

Each protocol session, i. e., each message exchange with at most two messages, is processed by one instance of $\mathcal{F}_{MX}$. The functionality is connected to the environment (or another IITM, depending on the overall system) with two pairs of tapes, one for the *client side* of the communication and one for the *server side*. $\mathcal{F}_{MX}$ is able to receive the request on the client side and transfer it to the server side, and to transfer the response vice versa. $\mathcal{F}_{MX}$ also defines how the adversary is able to interfere with the message transfer.

In contrast to the *one instance per session* pattern of $\mathcal{F}_{MX}$, the functionality $\mathcal{F}_{SM}$ is long-lived: For each identity, at most one instance of $\mathcal{F}_{SM}$ runs and manages this server's resources (and, in case passwords are used as explained below, the clients' passwords).

The enriching input functionality just forwards resources from the environment to the adversary. This does not make sense in the ideal functionality alone, but it is used for an additional feature in the implementation as explained in Section 4.3.1.2.

In the next section, we describe how $\mathcal{F}_{\mathrm{MX}}$ and $\mathcal{F}_{\mathrm{SM}}$ are parameterized to allow for different implementations. We then discuss our modeling of password-based security in Section 4.2.2. Next, in Section 4.2.3 and 4.2.4, we describe the functionality in more detail, first the interface for the environment as well as an example message flow for an intended use of the functionality, and second the interface for the adversary. The corruption mechanism, which allows the adversary to corrupt each instance of $\mathcal{F}_{\mathrm{MX}}$ either on the client side or on the server side (or on both sides), is described in Section 4.2.5. The actual functionalities are given in the Appendix B.1, a diagram of the states and steps of $\mathcal{F}_{\mathrm{MX}}$ is given in Figure B.1.

Note that in $\mathcal{F}_{\mathrm{S2ME}}$, we use the underlined version of $\mathcal{F}_{\mathrm{SM}}$ and the double underlined version of $\mathcal{F}_{\mathrm{MX}}$ and $\mathcal{F}_{\mathrm{EI}}$. Thus, the messages of $\mathcal{F}_{\mathrm{SM}}$ seen in Appendix B.1.2 are prefixed with the server's identity, while the messages of $\mathcal{F}_{\mathrm{MX}}$ and $\mathcal{F}_{\mathrm{EI}}$ in Appendices B.1.1 and B.1.3 are prefixed with both the server's identity as well as the client's identity. E. g., when sending a response from server s to client c using step (B.4), the environment would actually send a message of the form $(s, c, sid_{\mathrm{s}}, \text{Response}, p_{\mathrm{s}})$.

4.2.1. Parameters

The ideal functionality is parameterized such that it is flexible enough for different implementations, and thus, in Section 4.3, we are able to provide three different realizations along the lines of Section 2.4:

- The first realization guarantees the authenticity of the exchanged messages by using digital signatures; this is the adaptation of the protocol SA2ME-1 introduced in Section 2.5.1 and analyzed in Chapter 3.
- The second realization implements the protocol CSA2ME-1 from Section 2.5.2 and not only ensures authenticity of the messages, but also confidentiality of the payloads transferred. It uses digital signatures and hybrid encryption.
- The third realization, implementing the protocol PA2ME-1 from Section 2.5.3, guarantees the authenticity of the exchanged messages (but, as discussed below, in a weaker sense) under the (realistic) assumption that only the servers have signature keys usable for authenticating messages, while the clients use passwords to authenticate the requests.

To allow for this kind of flexibility, the ideal functionality has two parameters:

- The parameter *leak*: $\{1\}^* \times \{0,1\}^* \to \{0,1\}^*$ for $\mathcal{F}_{\mathrm{MX}}$ is called the *leakage algorithm* that describes what information about the payloads is visible to the adversary.

  In the realizations of SA2ME-1 and PA2ME-1, which offer no confidentiality, the leakage algorithm we use is the *full leakage* $leak_{\mathrm{full}}$ defined by $leak_{\mathrm{full}} \colon (x, y) \mapsto y$, modeling that the payloads are fully visible to the adversary. In the realization of CSA2ME-1, only partial information about the payloads leaks to the adversary, i. e., the length of the payload.

See Section 4.3.3 for details on leakage algorithms.

- The parameter *pw-auth* $\in \{\texttt{true}, \texttt{false}\}$ for $\mathcal{F}_{\text{MX}}$ and $\mathcal{F}_{\text{SM}}$ controls whether passwords are used for authentication. In that case, we provide the adversary with additional abilities, e. g., to guess passwords or to test whether the password used in a session is correct.

Note that these two parameters do not only allow the three realizations mentioned above, but imply at least a fourth version, see Section 2.4. The leakage parameter even allows additional versions, e. g., protocols that even hide the length of the payload if that is below some fixed value, or only hide parts of the payload.

4.2.2. Analyzing Password-Based Security

When analyzing password-based security in the IITM framework (or any other framework with asymptotic security definitions), the simplest approach would be to assume that passwords are chosen uniformly at random from some set that grows with the security parameter; as already mentioned in Section 2.1.3.6, this excludes realistic attack scenarios. In contrast, we abstain from making such assumptions, but instead try to realistically model different aspects of the problem along the lines of [CHK$^+$05].

First, we have to model how passwords are chosen. In our functionalities, we let the environment provide the passwords for clients and server as part of the input. This effectively means that we do not make assumptions about the probability distribution for passwords or the relation between passwords, e. g., of one user for different servers etc.

It also means that our modeling covers all possibilities what the adversary knows or learns from external sources (i. e., not our protocol): As the security definition quantifies over all environments and all adversaries, it includes the cases where the environment fully or partially (or not at all) cooperates with the adversary. This means that even if the adversary knows all or some passwords or if it can choose all or some passwords, the protocol stays secure in the sense defined by the ideal functionality.

Next, we have to model the realistic and unavoidable ability of the adversary to test if a certain password is correct: In realistic setups, the adversary can always send a request message to a server using the assumed password of a client, and then decide based on the server's reaction if the assumed password was correct. We model this by simply allowing the adversary to explicitly test if a password is correct, see Section 4.2.4 for an overview of the abilities of the adversary in our modeling.

But in addition to this unavoidable ability to test passwords online, i. e., interactively with a server, a naïve implementation could be vulnerable to an offline attack, where an adversary intercepts a message and then, in an offline phase (i. e., without communicating with the protocol's participants), tries to learn something about the password. For example, a simple protocol that sends a hash of the password in the clear would not be able to counter offline attacks as the adversary may, e. g., use a dictionary of common passwords, hash each of them and compare the result to the intercepted password

hash. Our ideal functionality, on the other hand, guarantees that the adversary does not receive any information about the password used in an uncorrupted session except its length and whether the password supplied by the environment is correct: Unless the functionality is corrupted, neither the password nor any value computed from it or depending on it (except its length and whether the password ist correct) are sent to the any other machine (except $\mathcal{F}_{\text{SM}}$).

Informally, this guarantees that the online attack is the "best" possible way for the adversary to learn something about the password, and that the adversary can only test passwords one at a time (i. e., testing a password does not yield information about any other password). We remark that the server can easily detect online guessing attacks and partially impede online guessing attacks[12], e. g., by limiting the number of (wrong) passwords that a user is allowed to try out in a certain time frame, or by locking accounts after too many failed authentication attempts occurred.

In addition to the above, our ideal functionality also provides the following guarantee: Even if an adversary knows a client's password, it is still not able to "break into" an uncorrupted instance of the $\mathcal{F}_{\text{MX}}$ functionality of that client, more precisely, the adversary may start new sessions if it knows a password, but once a client initiated a session, the adversary has no means—besides corruption—to change the request or the response such that the client accepts the response.

4.2.3. Regular Operation and the Interface for the Environment

First, we remark that our functionality can be employed by other IITM's to implement, e. g., high-level protocols which build upon secure two-round message exchange. Nevertheless, without loss of generality, in what follows we describe our functionality in a setting where it is directly used by the environment, as this setting is the one relevant for the security proofs later on.

The environment communicates with $\mathcal{F}_{\text{MX}}$ over two pairs of tapes, E^{c}_{MX} and E^{s}_{MX}, which handle the client side and the server side, respectively. $\mathcal{F}_{\text{SM}}$ is accessed by the environment on a pair of tapes named E_{SM}.

The intended use of the functionality to transfer payloads between a client c and a server s is roughly as follows, where we explain the steps and illustrate them by example messages:

0. The environment initializes the server s, i. e., starts an instance of $\mathcal{F}_{\text{SM}}$ for s (see step (B.14)); and, if passwords are used, the environment provides $\mathcal{F}_{\text{SM}}$ with a list of users and their passwords:

   ◦ $E_{\text{SM}} \to \mathcal{F}_{\text{SM}}\colon (s, \mathsf{Init}, [c \to pw, c' \to pw', \ldots])$

   The new instance of $\mathcal{F}_{\text{SM}}$ then informs the adversary of its initialization and waits for the adversaries approval (see next section); the list of users and their pass-

[12] Note that some of those mechanisms, while helping against online guessing attacks, open some possibility for denial-of-service attacks against users.

words is stored in $\mathcal{F}_{\text{SM}}$. The environment only has to initialize $\mathcal{F}_{\text{SM}}$ once per server identity s.

1. The environment supplies $\mathcal{F}_{\text{SM}}$ with the necessary resources to receive a message using step (B.15) in the form of a bit string of some length n_{s}:

   ◦ $E_{\text{SM}} \rightarrow \mathcal{F}_{\text{SM}}\colon (s, \text{Resources}, 1^{n_{\text{s}}})$

   The resources are stored in $\mathcal{F}_{\text{SM}}$ and the adversary is informed about the resources.

2. The environment starts a new session (and thus, a new instance of $\mathcal{F}_{\text{MX}}$) by sending the *request to send a request* on the client side (step (B.1)) and passing the request payload p_{c}. For this, the environment chooses a session number on the client side (sid_{c}), which is used in this session when $\mathcal{F}_{\text{MX}}$ communicates with E^{c}_{MX}. The environment also has to pass resources to $\mathcal{F}_{\text{MX}}$ in the form of a bit string of some length n_{c}, which enables $\mathcal{F}_{\text{MX}}$ to receive a response to this request. If passwords are used, the environment should provide the client's password pw (if passwords are not used, the environment may simply pass ε as password):

   ◦ $E^{\text{c}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}\colon (s, c, sid_{\text{c}}, \text{Request}, p_{\text{c}}, pw, 1^{n_{\text{c}}})$

   Upon receiving this request, $\mathcal{F}_{\text{MX}}$ generates random session numbers for communicating with the server side of the environment (sid_{s}) and the adversary (sid_{A}). Then, $\mathcal{F}_{\text{MX}}$ asks for the adversaries approval to send the request message (see next section). If the adversary grants the transfer, $\mathcal{F}_{\text{MX}}$ tries to obtain resources from $\mathcal{F}_{\text{SM}}$ (which $\mathcal{F}_{\text{SM}}$ provides if resources are available, step (B.16)) and test whether the password was correct (step (B.19)):

   ◦ $\mathcal{F}_{\text{MX}} \rightarrow \mathcal{F}_{\text{SM}}\colon (s, c, sid_{\text{s}}, \text{GetSession})$
   ◦ $\mathcal{F}_{\text{SM}} \rightarrow \mathcal{F}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Session}, 1^{n_{\text{s}}})$
   ◦ $\mathcal{F}_{\text{MX}} \rightarrow \mathcal{F}_{\text{SM}}\colon (s, c, sid_{\text{s}}, \text{Test}_{\text{internal}}, pw)$
   ◦ $\mathcal{F}_{\text{SM}} \rightarrow \mathcal{F}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Test}_{\text{internal}}, \texttt{true})$

3. After approval by the adversary, if enough resources are available on the server side and if the password is correct, the environment receives the request payload on the server side on tape E^{s}_{MX} (see step (B.3)):

   ◦ $\mathcal{F}_{\text{MX}} \rightarrow E^{\text{s}}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Request}, p_{\text{c}})$

4. Using sid_{s}, the environment is able to send a *request to send a response* on the server side (step (B.4)) containing a response payload p_{s}:

   ◦ $E^{\text{s}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Response}, p_{\text{s}})$

   If the session has been expired by the adversary, see below, an error message is sent back to E^{s}_{MX} (step (B.7)):

   ◦ $\mathcal{F}_{\text{MX}} \rightarrow E^{\text{s}}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Response}_{\text{Error}})$

   Otherwise, $\mathcal{F}_{\text{MX}}$ asks for the adversaries approval to send the message (see next section).

5. After approval by the adversary and if enough resources are available on the client side, the response is passed on to the environment on the client side (see step (B.5)):
   - $\mathcal{F}_{\mathrm{MX}} \to E^{\mathrm{c}}_{\mathrm{MX}}\colon (s, c, sid_{\mathrm{c}}, \mathsf{Response}, p_{\mathrm{s}})$

4.2.4. Attacks and the Interface for the Adversary

Ideally, one would want to transfer messages without interference by the adversary. Obviously, such an ideal functionality would not be realizable in a realistic way, as we assume that the adversary may at least block all network communication.

Therefore, we explicitly allow the adversary to influence the transfer in the following ways:

– Each time a message is passed from client to server or vice-versa, the adversary has to give its approval. But unless the session is corrupted (see Section 4.2.5), the adversary has no control over the contents: It may refuse to approve the message transfer, but it is not able to alter the contents of the message or redirect it etc..

  Therefore, in $\mathcal{F}_{\mathrm{MX}}$, before the request message is passed from the client to the server, the leakage of that message as well as the length of the password is sent to the adversary (see step (B.2)). If the corresponding instance of $\mathcal{F}_{\mathrm{MX}}$ is not corrupted (see next section), the adversary can only block the transfer (by not sending the approval message) or approve the transfer:
   - $\mathcal{F}_{\mathrm{MX}} \to A_{\mathrm{MX}}\colon (s, c, sid_A, \mathsf{Request}, leak(1^\eta, p_{\mathrm{c}}), |pw|, n_{\mathrm{c}})$
   - $A_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, sid_A, \mathsf{Request}_{\mathsf{OK}}, \varepsilon, \varepsilon)$

  Analogous messages are exchanged for the response message (step (B.5)):
   - $\mathcal{F}_{\mathrm{MX}} \to A_{\mathrm{MX}}\colon (s, c, sid_A, \mathsf{Response}, leak(1^\eta, p_{\mathrm{s}}))$
   - $A_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, sid_A, \mathsf{Response}_{\mathsf{OK}}, \varepsilon)$

  This models the assumed ability of a real-world adversary to block messages from being sent.

– When the request has been delivered to the environment on the server side, but the environment has not (yet) requested to send a response, the adversary may *expire the session* (step (B.6)):
   - $A_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, sid_A, \mathsf{Expire})$

  If, after expiration, the corresponding instance of $\mathcal{F}_{\mathrm{MX}}$ receives a request from the environment to deliver a response, it answers with an error message containing $\mathsf{Response}_{\mathsf{Error}}$ (see previous section).

  Allowing the adversary to expire sessions is necessary because we assume a realistic server has only a limited capacity to store session data. Note that explicit expiration is more than just allowing the adversary to block messages from reaching the client: If a session is expired on the server side, the server can respond to the environment with an error message. This is important, since in realistic situations, there also is a

difference between a server that responds with an error message to the environment and a server that tries, but unknowingly fails to deliver a message.

If passwords are used, the following additional abilities are modeled (also see Section 4.2.2 above):

- The adversary is allowed to test passwords (step (B.18)): It can send an identity c and a guess for this identities password pw to a server, which answers whether the password is correct:

  ◦ $A_{\mathrm{SM}} \to \mathcal{F}_{\mathrm{SM}}\colon (s, c, \mathrm{Test}, pw)$
  ◦ $\mathcal{F}_{\mathrm{SM}} \to A_{\mathrm{SM}}\colon (s, c, \mathrm{Test}, \mathtt{true})$

  The adversary can also ask a session of $\mathcal{F}_{\mathrm{MX}}$ whether the password provided by the environment and stored in $\mathcal{F}_{\mathrm{MX}}$ is correct (step (B.9)):

  ◦ $A_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, sid_A, \mathrm{Test})$
  ◦ $\mathcal{F}_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{SM}}\colon (s, c, \mathrm{Test}, pw)$
  ◦ $\mathcal{F}_{\mathrm{SM}} \to A_{\mathrm{SM}}\colon (s, c, \mathrm{Test}, \mathtt{true})$

- If the adversary knows the correct password pw of a client c, it may start a *server-only session*—an instance of $\mathcal{F}_{\mathrm{MX}}$ which only communicates with the environment on the server side, but not on the client side. This models the realistic ability of an adversary who knows a client's password to initiate (forged) sessions under the identity of this client without involving the client.

  The adversary sends the request to start a server-only session to $\mathcal{F}_{\mathrm{SM}}$ containing some payload p_{c}, see step (B.17) (where *cor* reflects whether the session is corrupted, see next section):

  ◦ $A_{\mathrm{SM}} \to \mathcal{F}_{\mathrm{SM}}\colon (s, \mathrm{Session}, c, cor, pw, p_{\mathrm{c}})$

  In this case, $\mathcal{F}_{\mathrm{SM}}$ starts an instance of $\mathcal{F}_{\mathrm{MX}}$ which stores the information that this is a server-only session, then generates sid_{s} and sid_A as above, informs the adversary of the session number sid_A and delivers the payload to the environment on the server side (step (B.8)):

  ◦ $\mathcal{F}_{\mathrm{SM}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, \mathrm{Session}, cor, p_{\mathrm{c}})$
  ◦ $\mathcal{F}_{\mathrm{MX}} \to A_{\mathrm{MX}}\colon (s, c, sid_A, \mathrm{Session})$
  ◦ $A_{\mathrm{MX}} \to \mathcal{F}_{\mathrm{MX}}\colon (s, c, sid_A, \mathrm{Session}_{\mathrm{OK}})$
  ◦ $\mathcal{F}_{\mathrm{MX}} \to E^{\mathrm{s}}_{\mathrm{MX}}\colon (s, c, sid_{\mathrm{s}}, \mathrm{Request}, p_{\mathrm{c}})$

  If the environment tries to respond to a server-only session, the response payload is delivered to the adversary, but not to the environment on the client side since there is no client involved in the session.

  In contrast to server-only sessions, we call sessions initiated by the environment *full sessions*.

4.2.5. Corruption

$\mathcal{F}_{\text{MX}}$ allows (partial) corruption by the adversary: The adversary may choose to corrupt either the client side or the server side, or both sides. Corrupting the client side implies that the adversary is able to act in the role of the client, more precisely, (i) read the payload that the environment wants to send to the server, even if the payload was confidential before, and (ii) manipulate the payload sent to the server. For the server side, the situation is analogously.

In more detail, the corruption interface is as follows:

1. The adversary can corrupt each instance of $\mathcal{F}_{\text{MX}}$ on both the client and server sides using step (B.10):
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Corrupt}, \text{c})$
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Corrupt}_{\text{OK}}, \text{c})$
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Corrupt}, \text{s})$
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Corrupt}_{\text{OK}}, \text{s})$
2. The adversary can ask each corrupted instance of $\mathcal{F}_{\text{MX}}$ to reveal the client's and the server's payloads as well as the client's password (see step (B.11)).
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Reveal}, \text{c})$
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Reveal}, \text{c}, p_{\text{c}}, pw)$
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Reveal}, \text{s})$
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Reveal}, \text{s}, p_{\text{s}}, pw)$

   Internally, $\mathcal{F}_{\text{MX}}$ ensures that, for each side, this is only done once per session using variables $revealed_{\text{c}}$ and $revealed_{\text{s}}$.
3. The adversary can manipulate the payloads of the request and response messages: Instead of just giving its approval to send a message in steps (B.2) or (B.5), it may pass its own payload into the sessions and analogously manipulate the password:
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Request}, leak(1^\eta, p_{\text{c}}), |pw|, n_{\text{c}})$
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Request}_{\text{OK}}, p'_{\text{c}}, pw')$
   - $\mathcal{F}_{\text{MX}} \to E^{\text{s}}_{\text{MX}}\colon (s, c, sid_{\text{s}}, \text{Request}, p'_{\text{c}})$

   Note that $\mathcal{F}_{\text{MX}}$ delivers p'_{c} instead of p_{c} (but only if pw' is the correct password, otherwise the message is not delivered).

   Analogous messages are exchanged for the response message:
   - $\mathcal{F}_{\text{MX}} \to A_{\text{MX}}\colon (s, c, sid_A, \text{Response}, leak(1^\eta, p_{\text{s}}))$
   - $A_{\text{MX}} \to \mathcal{F}_{\text{MX}}\colon (s, c, sid_A, \text{Response}_{\text{OK}}, p'_{\text{s}})$
   - $\mathcal{F}_{\text{MX}} \to E^{\text{c}}_{\text{MX}}\colon (s, c, sid_{\text{c}}, \text{Request}, p'_{\text{s}})$
4. If a full-session instance of $\mathcal{F}_{\text{MX}}$ is corrupted on the server side, the adversary can deliver a response to the client even before the delivery of the request message is approved (if enough resources are available, see step (B.5)):

- $A_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}$: $(s, c, sid_A, \text{Response}_{\text{OK}}, p'_{\text{s}})$
- $\mathcal{F}_{\text{MX}} \rightarrow E^{\text{c}}_{\text{MX}}$: $(s, c, sid_{\text{c}}, \text{Response}, p'_{\text{s}})$

Internally, $\mathcal{F}_{\text{MX}}$ ensures that this is only done once per session using a variable $replied_{\text{c}}$.

5. The environment can use step (B.12) to ask an instance (both on the client side and the server side) if it is corrupted:
   - $E^{\text{c}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}$: $(s, c, sid_{\text{c}}, \text{Corrupted})$
   - $\mathcal{F}_{\text{MX}} \rightarrow E^{\text{c}}_{\text{MX}}$: $(s, c, sid_{\text{c}}, \text{Corrupted}, \texttt{true})$
   - $E^{\text{s}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}$: $(s, c, sid_{\text{s}}, \text{Corrupted})$
   - $\mathcal{F}_{\text{MX}} \rightarrow E^{\text{s}}_{\text{MX}}$: $(s, c, sid_{\text{s}}, \text{Corrupted}, \texttt{true})$

   Note that this is even possible after the session has been expired etc., thus, the information whether a session was corrupted is never "lost".

6. The environment can provide resources for both sides of corrupted instances (see step (B.13)), and $\mathcal{F}_{\text{MX}}$ informs the adversary of the resources:
   - $E^{\text{c}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}$: $(s, c, sid_{\text{c}}, \text{Resources}, 1^n)$
   - $\mathcal{F}_{\text{MX}} \rightarrow A_{\text{MX}}$: $(s, c, sid_A, \text{Resources}, \text{c}, 1^n)$
   - $E^{\text{s}}_{\text{MX}} \rightarrow \mathcal{F}_{\text{MX}}$: $(s, c, sid_{\text{s}}, \text{Resources}, 1^{n'})$
   - $\mathcal{F}_{\text{MX}} \rightarrow A_{\text{MX}}$: $(s, c, sid_A, \text{Resources}, \text{s}, 1^{n'})$

   While this is not necessary in the ideal world alone, it is necessary to securely realize this functionality.

4.3. Realizing Secure Two-Round Message Exchange

In this section we describe three different realizations of $\mathcal{F}_{\text{S2ME}}$, see Table 4.1, where we use different sets of parameters for $\mathcal{F}_{\text{S2ME}}$.

To describe the realizations, we first introduce some prerequisites and functionalities defined elsewhere that we use in our functionalities. We then describe the implementation *signature authenticated* ($\mathcal{P}^{\text{SA}}_{\text{S2ME}}$) in detail and later point out the differences between $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ and the implementations *confidential, signature authenticated* ($\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$) as well as *password authenticated* ($\mathcal{P}^{\text{PA}}_{\text{S2ME}}$).

In Figure 4.4, we give an overview of the realizations including their connections to the environment and the adversary.[13]

4.3.1. Prerequisites and Used Functionalities

In this section we introduce auxiliary functionalities that will be used by the protocol functionalities later on. Examples of possible message transfers with these functionalities can be found in Section 4.3.2 and subsequent sections.

[13] Note that this is a simplified and generalized illustration (for example, it includes encryption functionalities although $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ does not contain those).

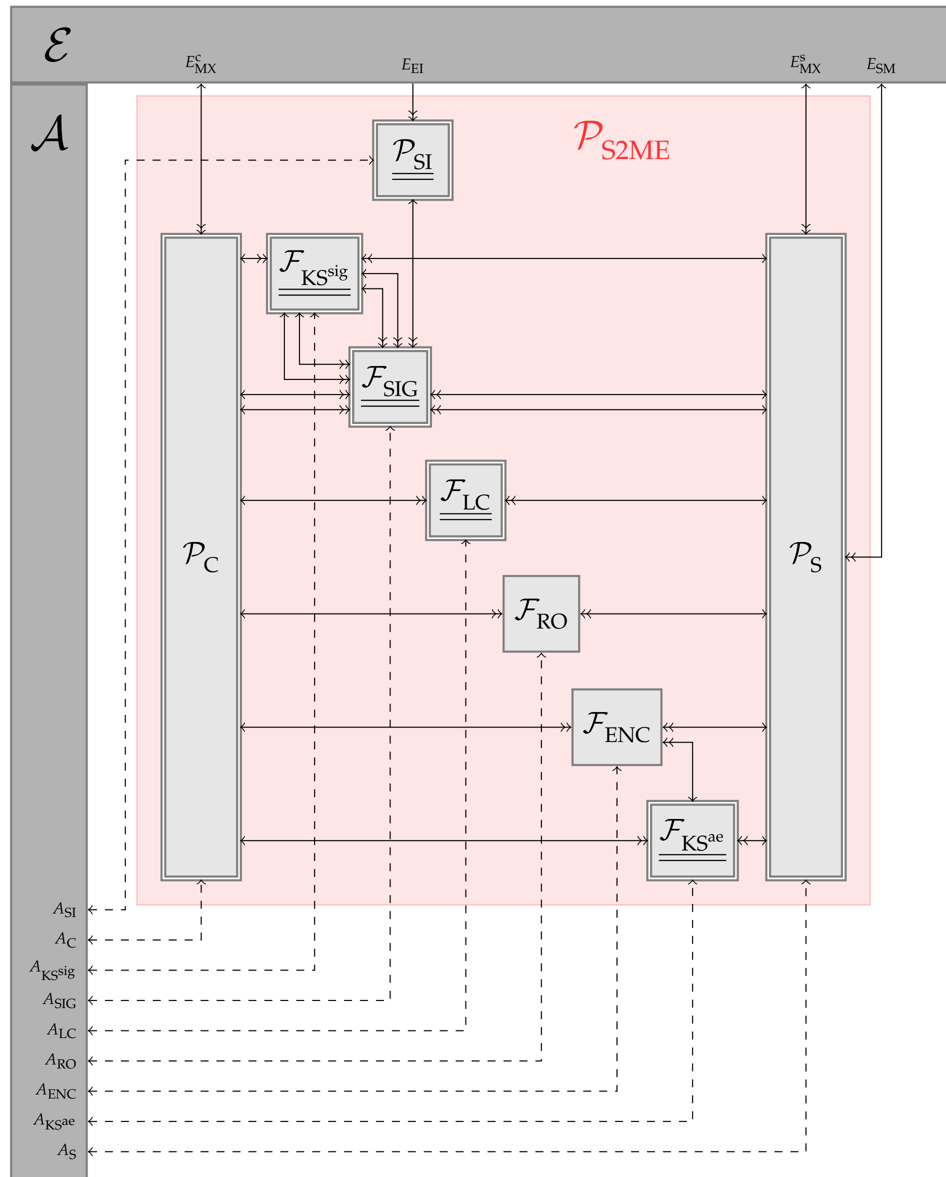

Figure 4.4.: An overview of a realization $\mathcal{P}_{S2ME}$ connected to the environment and the adversary

Name	Protocol	Ideal Functionality	Realization
SA	SA2ME-1 (see Section 2.5.1)	$\mathcal{F}_{\text{S2ME}}(\mathit{leak}_{\text{full}}, \texttt{false})$	$\mathcal{P}^{\text{SA}}_{\text{S2ME}}$
CSA	CSA2ME-1 (see Section 2.5.2)	$\mathcal{F}_{\text{S2ME}}(\mathit{leak}_{\text{length}}, \texttt{false})$	$\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$
PA	PA2ME-1 (see Section 2.5.3)	$\mathcal{F}_{\text{S2ME}}(\mathit{leak}_{\text{full}}, \texttt{true})$	$\mathcal{P}^{\text{PA}}_{\text{S2ME}}$

Table 4.1.: Different realizations of $\mathcal{F}_{\text{S2ME}}$ (The leakage algorithms $\mathit{leak}_{\text{full}}$ and $\mathit{leak}_{\text{length}}$ are defined in Section 4.3.1.4.)

4.3.1.1. The Signature Functionality $\mathcal{F}_{\text{SIG}}$

For modeling digital signatures, we use the functionality $\mathcal{F}_{\text{SIG}}$ from [KT08a, KT08b], which can be implemented using any EUF-CMA secure signature scheme.

We give a brief overview of the functionality; for details of the ideal functionality and securely realizations, we refer the reader to [KT08b]. $\mathcal{F}_{\text{SIG}}$ consists of two parts, a signature functionality and a verification functionality.

Upon initialization by the user, the signature functionality returns a public key pk^{sig} to the user. Now, the user is able to send messages to the signature functionality, which then returns a signature of that message. The verification functionality is called with an additional id (so that each party using the verification functionality can use its own copy of the verification functionality) and can be asked to verify the signature of a fiven a message under a given public key.

Internally, the functionality lets the adversary choose the algorithms for signature generation and verification, which are later executed each time a user instructs the functionality to sign messages or verify signatures. But the functionality guarantees that 1. any signature generated by the functionality is accepted as valid for the public key used in the functionality, and 2. if the correct key is used for verification and the functionality is not corrupted, the verification does only return "true" if the signature has really been generated by the signature functionality; the latter is guaranteed by keeping a list H of signatures that have been generated by the functionality.

$\mathcal{F}_{\text{SIG}}$ has three parameters:

- $\mathcal{T}_{\text{sig}}$ is a set of tapes connecting to functionalities that are allowed to sign messages,
- $\mathcal{T}_{\text{ver}}$ is a set of tapes connecting to functionalities that are allowed to verify messages, and
- p_{sig} is a polynomial used to bound the size and runtime of the keys and algorithms used by $\mathcal{F}_{\text{SIG}}$.

In what follows, we use $\mathcal{F}_{\text{SIG}}(p_{\text{sig}})$ as an abbreviation for $\mathcal{F}_{\text{SIG}}(\mathcal{T}_{\text{sig}}, \mathcal{T}_{\text{ver}}, p_{\text{sig}})$ by fixing the following sets of (endpoints of) tapes for the rest of the thesis:

$$\mathcal{T}_{\text{sig}} = \{\text{C}_{\text{sig}}, \text{S}_{\text{sig}}, \text{SI}_{\text{sig}}, \text{KS}^{\text{sig}}_{\text{sig}}\}\ , \qquad \mathcal{T}_{\text{ver}} = \{\text{C}_{\text{ver}}, \text{S}_{\text{ver}}, \text{SI}_{\text{ver}}, \text{KS}^{\text{sig}}_{\text{ver}}\}\ . \tag{4.5}$$

Note that $\mathcal{F}_{\text{SIG}}$ allows dynamic corruption, the adversary may separately corrupt each instance of the signature and verification functionality at any point.

4.3.1.2. The Signature Interface Functionality $\mathcal{P}_{\text{SI}}$

We give the adversary access to the signature functionality $\mathcal{F}_{\text{SIG}}$ through the signature interface functionality $\mathcal{P}_{\text{SI}}$ defined in Appendix B.2.9: The functionality allows the adversary to sign any bit string that does not have the format of a protocol message. As explained in Section 2.3.2.3, this models that our protocol does not have exclusive access to the keys used to sign the messages. For example, the same key can be used to sign a protocol message and parts of the payload contained in that message.

The functionality $\mathcal{P}_{\text{SI}}$ accepts requests from the adversary to (i) sign messages that do not have the format of protocol messages and (ii) verify arbitrary signatures. The restriction that the adversary may not sign bit strings that have the format of a protocol message is implemented by the parameter *except*, see below.

In our realizations, the signature interface functionality appears in the scope of a bang in the multi-user multi-session version, effectively meaning that the adversary has access to all keys used in the protocol. As the signature interface needs resources from the environment to sign messages for the adversary, it has an enriching input tape E_{EI}. Its counterpart in the ideal system $\mathcal{F}_{\text{S2ME}}$ is a tape in the enriching input functionality EI.

Exception Sets We make exceptions to the adversaries access to ensure that the adversary cannot simply sign protocol messages and thus fake requests or responses without corrupting keys.

To this end, we define the three exception functions $except_{\text{SA2ME}-1}$, $except_{\text{CSA2ME}-1}$, and $except_{\text{PA2ME}-1}$, each of the type $\{0,1\}^* \rightarrow \{\texttt{true}, \texttt{false}\}$, used later on in our realizations for the parameter *except*.

- The exception function $except_{\text{PA2ME}-1}$ returns `true` if the input is of the form $(\mathsf{From}\colon \cdot, \mathsf{To}\colon \cdot, \mathsf{Ref}\colon \cdot, \mathsf{Body}\colon \cdot)$, and `false` otherwise.
- The exception function $except_{\text{SA2ME}-1}$ returns `true` if the input is of the form $(\mathsf{From}\colon \cdot, \mathsf{To}\colon \cdot, \mathsf{MsgID}\colon \cdot, \mathsf{Time}\colon \cdot, \mathsf{Body}\colon \cdot)$ or if $except_{\text{PA2ME}-1}$ returns `true` on the same input, and `false` otherwise.
- The exception function $except_{\text{CSA2ME}-1}$ returns `true` if the input is of the form $(\mathsf{From}\colon \cdot, \mathsf{To}\colon \cdot, \mathsf{MsgID}\colon \cdot, \mathsf{Time}\colon \cdot, \mathsf{Key}\colon \cdot, \mathsf{Body}\colon \cdot)$ or if $except_{\text{PA2ME}-1}$ returns `true` on the same input, and `false` otherwise.

4.3.1.3. The Signature Key Store Functionality $\mathcal{F}_{\text{KS}^{\text{sig}}}$

To coordinate how different IITM's access a single instance of the signature functionality, we define the ideal functionality of a signature key store ($\mathcal{F}_{\text{KS}}$) in Appendix B.2.7, which allows clients, servers, and the signature interface functionality to retrieve

trusted keys as well as the corruption status of keys. To be able to distribute the public keys, $\mathcal{F}_{\text{KS}}$ also initializes the instances of the signature functionality. The particular form of this functionality is due to the fact that we want to use $\mathcal{F}_{\text{SIG}}$ from [KT08a] as is.

4.3.1.4. The Encryption Functionality $\mathcal{F}_{\text{ENC}}$

A realistic modeling of encryption in the IITM framework is a challenging task, as hybrid encryption is often used for large plaintexts (see Section 2.1.3.4). We use the functionalities from [KT09a], which are equipped to handle asymmetric and symmetric encryption schemes as well as hybrid encryption, to model both our usage of asymmetric encryption (for PA2ME-1) and hybrid encryption (for CSA2ME-1).

The encryption functionality we use is composed of three parts: 1. $\mathcal{F}_{\text{pke}}$ is an ideal functionality that can be realized by a secure public key encryption scheme, 2. $\mathcal{F}^{\text{unauth}}_{\text{ltsenc}}$ is an ideal functionality modeling unauthenticated symmetric encryption with long-term keys[14], 3. $\mathcal{F}^{\text{unauth}}_{\text{senc}}$ is the ideal functionality that provides both unauthenticated symmetric encryption with short-term keys as well as an interface to the two functionalities above, which, in case of $\mathcal{F}_{\text{pke}}$, can be used to model hybrid encryption.

In the above, "unauthenticated" refers to the fact that, alternatively, the symmetric encryption functionalities in [KT09a] offers authenticated symmetric encryption, i. e., the guarantee that a valid ciphertext cannot be created without knowledge of the symmetric key. In contrast, for our purpose, the confidentiality guarantees suffice, i. e., we can assume that the adversary is able to create valid ciphertexts for any key without knowledge of that key.

Public Key and Symmetric Encryption We sketch how $\mathcal{F}_{\text{pke}}$ and $\mathcal{F}^{\text{unauth}}_{\text{senc}}$ operate, for details we refer the reader to [KT09a, KT09b].

Upon initialization by a user, $\mathcal{F}_{\text{pke}}$ lets the adversary specify algorithms for encryption, decryption and key generation, and the latter one is used to generate a key which is stored in the functionality and sent to the user.

The user then is able to request the encryption of a message and provide a public key for that encryption. If the key provided matches the one stored in the functionality, the functionality uses a *leakage algorithm* (see next section) to compute the leakage of the plaintext and then encrypt the leakage using the stored key and the algorithm provided by the adversary. Then, if the decryption algorithm is able to decrypt the ciphertext and return the leakage as expected, the original message together with the ciphertext is stored in a list *decTable* in the functionality, and the ciphertext is returned to the user. Thus, even if the encryption algorithm provided by the adversary leaks information (or does not encrypt at all), the functionality guarantees that the leakage algorithm can be used to bound how much information about the plaintext is leaked. If, on the other hand, the key provided by a user when requesting to encrypt a message is not the one

[14] Note that $\mathcal{F}^{\text{unauth}}_{\text{ltsenc}}$ is not used throughout this thesis but still included here as $\mathcal{F}^{\text{unauth}}_{\text{senc}}$ and the theorems in [KT09a], which are used below, include $\mathcal{F}^{\text{unauth}}_{\text{ltsenc}}$.

stored in the functionality, nothing is guaranteed, i. e., the algorithm provided by the adversary is directly used to encrypt the given message, and the ciphertext is returned to the user.

Now if a user requests to decrypt a ciphertext (and the functionality is not corrupted), the list *decTable* is used to lookup the plaintext corresponding to the ciphertext, and the plaintext is returned if there is exactly one entry in the list for the given ciphertext. Otherwise, the decryption algorithm provided by the adversary is used if there is no matching ciphertext in *decTable*, or decryption may fail (the functionality returns $\perp$) if multiple plaintexts are found in the list for the given ciphertext. In this way, the functionality guarantees that if the correct key is used for encryption and if no ciphertexts collide, decryption of a ciphertext returned by the functionality is successful.

Similarly to above, upon initialization by the user, $\mathcal{F}_{\text{senc}}^{\text{unauth}}$ lets the adversary specify encryption and decryption algorithms. Again, the encryption algorithm receives only the leakage of the plaintext that the user wants to encrypt, and a list *decTable* is used to allow for decryption.

But contrary to above, key management is an important part of $\mathcal{F}_{\text{senc}}^{\text{unauth}}$: For reasons explained in 4.5.4, the functionality does not hand out the symmetric keys to the user, but instead uses pointers to keys: When the user requests the functionality to generate a key, the functionality only returns a pointer to a freshly generated key that is stored inside the functionality. This user can then request the functionality to encrypt plaintexts or decrypt ciphertexts using a pointer to a key.

The functionality also supports the encryption of keys: The user can include the pointer to a key in the plaintext sent to the functionality, and the functionality replaces the pointer by the actual key before passing the plaintext to the leakage and encryption algorithms; and after decrypting a ciphertext received from the user, if the plaintext contains a key, that key is replaced by a pointer before the plaintext is returned to the user.

We note that $\mathcal{F}_{\text{ENC}}$ only allows static corruption, i. e., if a key is generated in $\mathcal{F}_{\text{pke}}$ or $\mathcal{F}_{\text{senc}}^{\text{unauth}}$, the adversary has to decide at that point if it wishes to corrupt that key; changing that decision later on is not possible. When a key is corrupted, both $\mathcal{F}_{\text{pke}}$ and $\mathcal{F}_{\text{senc}}^{\text{unauth}}$ do not use the leakage algorithm, but pass the given plaintext to the encryption algorithm directly.

Leakage Algorithms Any encryption scheme that is equipped to handle plaintexts of arbitrary lengths cannot prevent the leakage of information: From any ciphertext, it is at least possible to deduce some restriction on the length of the corresponding plaintext. Therefore, [KT09a] uses so-called *leakage algorithms* to precisely define how much information an encryption scheme leaks.

Informally, a leakage algorithm takes as input the security parameter and a bit string and returns a bit string—for a formal definition see [KT09a, KT09b]. A leakage algorithm that *leaks exactly the length of a message* is a leakage algorithm such that 1. for all valid input bitstrings, the algorithm returns a bitstring of the same length as the input

string, and 2. for any two valid input bitstrings of equal length, the output distributions of the algorithm for both inputs are the same.

Besides $\mathcal{F}_{\text{ENC}}$, our ideal functionality $\mathcal{F}_{\text{MX}}$ also takes a leakage algorithm as a parameter, for this purpose we define the following leakage algorithms:

- *Full leakage:* $\mathit{leak}_{\text{full}}(x,y)$ returns y.
- *Length leakage:* $\mathit{leak}_{\text{length}}(x,y)$ returns an element chosen uniformly at random from $\{0,1\}^{|y|}$.

Note that $\mathit{leak}_{\text{length}}$is an example for a leakage algorithm that leaks exactly the length of a message.

Notation For the rest of this thesis, we fix the following sets of tapes:

$$\mathcal{T}_{\text{ENC}} = (\mathcal{T}_{\text{users}}, T_{\text{adv}}) \quad \text{with} \quad \mathcal{T}_{\text{users}} = \{\text{C}, \text{S}, \text{KS}^{\text{ae}}\} \ , \tag{4.6}$$
$$\mathcal{T}^{\text{lt}}_{\text{ENC}} = (\mathcal{T}^{\text{lt}}_{\text{users}}, T^{\text{lt}}_{\text{adv}}) \quad \text{with} \quad \mathcal{T}^{\text{lt}}_{\text{users}} = \{T^{\text{lt}} \mid T \in \mathcal{T}_{\text{users}}\} \ , \tag{4.7}$$
$$\mathcal{T}^{\text{pke}}_{\text{ENC}} = (\mathcal{T}^{\text{pke}}_{\text{users}}, T^{\text{pke}}_{\text{adv}}) \quad \text{with} \quad \mathcal{T}^{\text{pke}}_{\text{users}} = \{T^{\text{pke}} \mid T \in \mathcal{T}_{\text{users}}\} \ . \tag{4.8}$$

The encryption functionalities use polynomials p_{st}, p_{lt}, and p_{ae} to bound the algorithms executed therein, as well as a leakage algorithm to model how much information a ciphertext leaks. Now, we define a system of IITM's called $\mathcal{F}_{\text{ENC}}$ composed of the symmetric, symmetric long-term and public key encryption functionalities defined in [KT09a]:

$$\mathcal{F}_{\text{ENC}}(\mathit{leak}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) = \\ \mathcal{F}^{\text{unauth}}_{\text{senc}}(p_{\text{st}}, \mathit{leak}, \mathcal{T}_{\text{ENC}}) \mid !\mathcal{F}^{\text{unauth}}_{\text{ltsenc}}(p_{\text{lt}}, \mathit{leak}, \mathcal{T}^{\text{lt}}_{\text{ENC}}) \mid \underline{!\mathcal{F}_{\text{pke}}(p_{\text{ae}}, \mathit{leak}, \mathcal{T}^{\text{pke}}_{\text{ENC}})} \ . \tag{4.9}$$

4.3.1.5. The Encryption Key Store Functionality $\mathcal{F}_{\text{KS}^{\text{ae}}}$

Analogously to the key store for signature keys in Section 4.3.1.3, this key store, defined in Appendix B.2.8, manages access to public keys used for public key encryption.

4.3.1.6. The Random Oracle Functionality $\mathcal{F}_{\text{RO}}$

As mentioned in Section 2.1.3.5, hash functions are hard to analyze formally, thus, one often uses the random oracle model in spite of its weaknesses.

We also use a random oracle to model hash functions, although it cannot be securely realized as shown in [CGH04]. We define a simple random oracle functionality $\mathcal{F}_{\text{RO}}$ for the IITM framework, see Appendix B.2.12. It models a random oracle that can be accessed by clients and servers, but also by the adversary. We note that a similar functionality was defined in [HMQ04] for the universal composition framework.

In our functionality, to fulfill the resource restrictions, we have to limit the adversary's access, so we allow the adversary one call to the random oracle per call of another machine (client or server). This is no real restriction as the environment is free to start an arbitrary number of clients, and the security definition quantifies over all environments, i. e., also over those environments that cooperate with the adversary to start as many clients as necessary.

In the rest of this chapter, we refer to calls to the oracle as "hashing", e. g., we say that an IITM hashes a value if it sends the value to the random oracle functionality. In addition, our proofs do not rely on any properties of the random oracle functionality directly, but instead refer to the three security properties specified in Section 2.1.3.5 which are guaranteed by the random oracle functionality, namely preimage resistance, second preimage resistance, and collision resistance.

4.3.1.7. The Local Clock Functionality $\mathcal{F}_{\text{LC}}$

The local clock functionality $\mathcal{F}_{\text{LC}}$, see Appendix B.2.11, models a clock that can be controlled by the adversary, but with two limitations: First, the clock remains monotonous, i. e., the adversary cannot decrease the value of the clock, and second, the adversary is not called each time a participant uses its clock, i. e., the adversary cannot block the access to the clock functionality.

4.3.2. Signature-Authenticated Two-Round Message Exchange

The system of IITM's that later implements $\mathcal{F}_{\text{S2ME}}(leak_{\text{full}}, \texttt{false})$ is defined for any polynomial p_{sig} by

$$\mathcal{P}^{\text{SA}}_{\text{S2ME}}(p_{\text{sig}}) = !\mathcal{P}^{\text{SA}}_{\text{C}} \mid !\mathcal{P}^{\text{SA}}_{\text{S}} \mid \underline{\underline{!\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid \underline{\underline{!\mathcal{P}_{\text{SI}}(except_{\text{SA2ME}-1})}} \mid \underline{\underline{!\mathcal{F}_{\text{LC}}}} \mid \underline{\underline{!\mathcal{F}_{\text{SIG}}(p_{\text{sig}})}} \tag{4.10}$$

On the right hand side, the client and server functionalities $\mathcal{P}^{\text{SA}}_{\text{C}}$ and $\mathcal{P}^{\text{SA}}_{\text{S}}$, see Appendices B.2.1 and B.2.4, implement the client's part and the server's part of the protocol, respectively. They are described in more detail below, and we illustrate the functionalities by giving examples of possible sequences of message exchanged between the ITIM's.

4.3.2.1. The Client Functionality $\mathcal{P}^{\text{SA}}_{\text{C}}$

The client functionality, see Appendix B.2.1, uses one IITM instance per protocol session, therefore, the client is mainly defined by only two steps, forwarding the request from the environment to the network (i. e., the adversary), and forwarding the response vice versa.

Assume that the environment instructs the client functionality running under identity c to send a payload p_{c} to a server s using password pw and local session id sid_{c}, and to allocate n_{c} resources for the reply. Before sending the request over the network in step (B.21), the client generates a nonce r, retrieves its local time t from the local clock

functionality, constructs the message $m_\text{c} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_\text{c})$ and signs the message, which is then sent over the network, i. e., to the adversary:

- $E^\text{c}_\text{MX} \to \mathcal{P}^\text{SA}_\text{C}\colon (s, c, sid_\text{c}, \mathsf{Request}, p_\text{c}, pw, 1^{n_\text{c}})$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_\text{LC}\colon (c, (\mathsf{C}, s, r), \mathsf{GetTime})$
- $\mathcal{F}_\text{LC} \to \mathcal{P}^\text{SA}_\text{C}\colon (c, (\mathsf{C}, s, r), \mathsf{Time}, t)$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_{\text{KS}^\text{sig}}\colon (c, (\mathsf{C}, s, r), \mathsf{GetKey})$
- $\mathcal{F}_{\text{KS}^\text{sig}} \to \mathcal{P}^\text{SA}_\text{C}\colon (c, (\mathsf{C}, s, r), \mathsf{PublicKey}, pk^\text{sig}_\text{c})$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_\text{SIG}\colon (c, (\mathsf{C}, s, r), \mathsf{Sign}, m_\text{c})$
- $\mathcal{F}_\text{SIG} \to \mathcal{P}^\text{SA}_\text{C}\colon (c, (\mathsf{C}, s, r), \mathsf{Signature}, \sigma_\text{c})$
- $\mathcal{P}^\text{SA}_\text{C} \to A_\text{C}\colon (m_\text{c}, \sigma_\text{c})$

Upon receival of a potential response message $(m_\text{s}, \sigma_\text{s})$ with $m_\text{s} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon p_\text{s})$ in step (B.22), the client checks if it has enough resources to process the response (and halts if that is not the case), checks the server's signature σ_s (by retrieving the server's key pk^sig_s, initializing the verifier and then requesting it to verify the signature) and, if successful, forwards the response to the environment:

- $A_\text{C} \to \mathcal{P}^\text{SA}_\text{C}\colon (m_\text{s}, \sigma_\text{s})$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_{\text{KS}^\text{sig}}\colon (s, (\mathsf{S}, c, r), \mathsf{GetKey})$
- $\mathcal{F}_{\text{KS}^\text{sig}} \to \mathcal{P}^\text{SA}_\text{C}\colon (s, (\mathsf{S}, c, r), \mathsf{PublicKey}, pk^\text{sig}_\text{s})$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_\text{SIG}\colon (s, (\mathsf{S}, c, r), \mathsf{C}, \mathsf{Init})$
- $\mathcal{F}_\text{SIG} \to \mathcal{P}^\text{SA}_\text{C}\colon (s, (\mathsf{S}, c, r), \mathsf{C}, \mathsf{Inited})$
- $\mathcal{P}^\text{SA}_\text{C} \to \mathcal{F}_\text{SIG}\colon (s, (\mathsf{S}, c, r), \mathsf{Verify}, m_\text{s}, \sigma_\text{s}, pk^\text{sig}_\text{s})$
- $\mathcal{F}_\text{SIG} \to \mathcal{P}^\text{SA}_\text{C}\colon (s, (\mathsf{S}, c, r), \mathsf{Verified}, \mathtt{true})$
- $\mathcal{P}^\text{SA}_\text{C} \to E^\text{c}_\text{MX}\colon (s, c, sid_\text{c}, \mathsf{Response}, p_\text{s})$

If the environment questions whether the client is corrupted, step (B.24), it answers `true` if either its own signature scheme or the verifier that the server uses to verify the validity of this client's signature are corrupted (both conditions are checked directly by asking the functionalities whether they are corrupted). Resources that the environment provides for corruption are passed on to the signature scheme, see step (B.23).

4.3.2.2. The Server Functionality $\mathcal{P}^\text{SA}_\text{S}$

Contrary to the client functionality, the server functionality in Appendix B.2.4 is long-lived in that only one instance of the server functionality is run per identity. This allows the functionality, e. g., to store the list of previously seen message id's. This has two consequences:

1. The functionality has to cope with the situation that while processing a message (e. g., checking its signature), another message from a different session of the protocol may arrive. In this case, our server implementation cancels the processing of the first message and allows the second message to be processed.

2. The functionality has to keep track of the sessions that is has processed, amongst other reasons because each of these sessions can be corrupted separately. Therefore, the server manages two lists, L and L_{cor}, of which the second one is only used for corruption.

   Each tuple in these lists has four components: a) The timestamp t of the message, b) the nonce r sent by the client, c) the client's identity c, and d) the session id sid_s used when the server communicates with the environment.

   If a message is received from a client and accepted (see step (B.39)), the server inserts a tuple for that message in both L and L_{cor}. If L exceeds the capacity cap of that server, it removes tuples from L (but not from L_{cor}) until L is small enough. If a reply is sent out for one of the requests with an entry in L, this entry is updated by replacing sid_s with ε to ensure that the environment cannot send a second reply within the same session.

   Note that while L complies with the memory restriction, in contrast, L_{cor} grows monotonously and hence may violate the memory restriction, but as this list is only used for keeping track of the corruption status of sessions, it would not be stored in a real implementation.

As a first step (B.33), the server has to be initialized by the environment. The server then asks the adversary to set the protocol's parameters, namely the capacity cap and the time tolerance tol^+; it also initializes the local clock to get the initial time t_s:

- $E_\mathrm{SM} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (s, \mathsf{Init}, [c \to pw, c' \to pw', \ldots])$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to A_\mathrm{S}\colon (s, \mathsf{GetParameters})$
- $A_\mathrm{S} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (s, \mathsf{Parameters}, \mathsf{cap}, \mathsf{tol}^+)$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to \mathcal{F}_\mathrm{LC}\colon (s, \mathsf{S}, \mathsf{GetTime})$
- $\mathcal{F}_\mathrm{LC} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (s, \mathsf{S}, \mathsf{Time}, t_\mathrm{s})$

If a message $m_\mathrm{c} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t_\mathrm{c}, \mathsf{Body}\colon p_\mathrm{c})$ is received from the network, the server 1. requests the client's signature key $pk^\mathrm{sig}_\mathrm{c}$ from the key store, see step (B.35), 2. gets the local time from the time functionality, step (B.36), 3. initializes the verifier and verifies the client's signature, steps (B.37) and (B.38), 4. checks the protocol's conditions for accepting messages, and if everything is in order, accepts the message, modifies the list L according to the protocol (see above) and then forwards the message to the environment, see step (B.39):

- $A_\mathrm{S} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (m_\mathrm{c}, \sigma_\mathrm{c})$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to \mathcal{F}_{\mathrm{KS}^\mathrm{sig}}\colon (c, (\mathsf{C}, s, r), \mathsf{GetKey})$
- $\mathcal{F}_{\mathrm{KS}^\mathrm{sig}} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (c, (\mathsf{C}, s, r), \mathsf{PublicKey}, pk^\mathrm{sig}_\mathrm{c})$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to \mathcal{F}_\mathrm{LC}\colon (s, \mathsf{S}, \mathsf{GetTime})$
- $\mathcal{F}_\mathrm{LC} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (s, \mathsf{S}, \mathsf{Time}, t_\mathrm{s})$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to \mathcal{F}_\mathrm{SIG}\colon (c, (\mathsf{C}, s, r), \mathsf{C}, \mathsf{Init})$
- $\mathcal{F}_\mathrm{SIG} \to \mathcal{P}^\mathrm{SA}_\mathrm{S}\colon (c, (\mathsf{C}, s, r), \mathsf{C}, \mathsf{Inited})$
- $\mathcal{P}^\mathrm{SA}_\mathrm{S} \to \mathcal{F}_\mathrm{SIG}\colon (c, (\mathsf{C}, s, r), \mathsf{Verify}, m_\mathrm{c}, \sigma_\mathrm{c}, pk^\mathrm{sig}_\mathrm{c})$

- $\mathcal{F}_{\text{SIG}} \to \mathcal{P}_{\text{S}}^{\text{SA}}: (c, (\text{C}, s, r), \text{Verified}, \texttt{true})$
- $\mathcal{P}_{\text{S}}^{\text{SA}} \to E_{\text{MX}}^{\text{s}}: (s, c, sid_{\text{s}}, \text{Request}, p_{\text{c}})$

If the server receives a response payload p_{s} for a session number sid_s from the environment, it first retrieves information from L about that session. If there is no matching entry in L with session number sid_s, an error message is sent to the environment meaning that it is not or no longer possible to send a response in that session:

- $E_{\text{MX}}^{\text{s}} \to \mathcal{P}_{\text{S}}^{\text{SA}}: (s, c, sid'_{\text{s}}, \text{Response}, p_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{SA}} \to E_{\text{MX}}^{\text{s}}: (s, c, sid'_{\text{s}}, \text{Response}_{\text{Error}})$

If instead a matching entry is found in L containing the nonce r, the server 1. updates the list L by replacing the session id sid_{s} with ε, constructs the response message $m_{\text{s}} = (\text{From}: c, \text{To}: s, \text{Ref}: r, \text{Body}: p_{\text{s}})$, and requests the signature key, step (B.40), 2. signs the response message, step (B.41), and 3. sends the signed response message to the network (i. e., the adversary), step (B.42):

- $E_{\text{MX}}^{\text{s}} \to \mathcal{P}_{\text{S}}^{\text{SA}}: (s, c, sid_{\text{s}}, \text{Response}, p_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{SA}} \to \mathcal{F}_{\text{KS}^{\text{sig}}}: (s, (\text{S}, c, r), \text{GetKey})$
- $\mathcal{F}_{\text{KS}^{\text{sig}}} \to \mathcal{P}_{\text{S}}^{\text{SA}}: (s, (\text{S}, c, r), \text{PublicKey}, pk_{\text{s}}^{\text{sig}})$
- $\mathcal{P}_{\text{S}}^{\text{SA}} \to \mathcal{F}_{\text{SIG}}: (s, (\text{S}, c, r), \text{Sign}, m_{\text{s}})$
- $\mathcal{F}_{\text{SIG}} \to \mathcal{P}_{\text{S}}^{\text{SA}}: (s, (\text{S}, c, r), \text{Signature}, \sigma_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{SA}} \to A_{\text{S}}: (m_{\text{s}}, \sigma_{\text{s}})$

When the adversary resets the server using step (B.43), the list L is cleared and any processing of a request or response is stopped.

Similar to the client, the environment can ask whether a session is corrupted on the server side. Therefore, as explained above, the server keeps a list L_{cor} of all sessions, even those that were deleted from L. For each session, if the environment calls step (B.46), the server answers `true` if either its own signature scheme used in that session or the verifier that the client uses in this session to verify the server's signature are corrupted (again, this is checked by asking the corresponding instances of the functionalities whether they are corrupted). Resources that the environment provides for corruption are passed on to the signature scheme used in that session, see step (B.45).

4.3.3. Confidential Signature-Authenticated Two-Round Message Exchange

The system of IITM's implementing $\mathcal{F}_{\text{S2ME}}(leak_{\text{length}}, \texttt{false})$ is defined for any leakage algorithm *leak* and any polynomials p_{sig}, p_{st}, p_{lt}, and p_{ae} by

$$\mathcal{P}_{\text{S2ME}}^{\text{CSA}}(leak, p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) = \\ !\mathcal{P}_{\text{C}}^{\text{CSA}} \mid !\mathcal{P}_{\text{S}}^{\text{CSA}} \mid \underline{\underline{!\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid \underline{\underline{!\mathcal{F}_{\text{KS}^{\text{ae}}}}} \mid \underline{\underline{!\mathcal{P}_{\text{SI}}(except_{\text{CSA2ME}-1})}} \mid \underline{\underline{!\mathcal{F}_{\text{LC}}}} \mid \\ \underline{\underline{!\mathcal{F}_{\text{SIG}}(p_{\text{sig}})}} \mid \mathcal{F}_{\text{ENC}}(leak, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) \ . \quad (4.11)$$

This implementation is similar to the one in Section 4.3.2, so we mainly point out the differences to that section below.

4.3.3.1. The Client Functionality $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}$

The client functionality is presented in Appendix B.2.2. When asked to send a request payload p_{c} in step (B.25), the client first generates a nonce r and gets the local time t as above. Then, the client obtains both a pointer ptr to a fresh key k generated for the symmetric encryption part of $\mathcal{F}_{\mathrm{ENC}}$ as well as the public encryption key pk_s^{ae} of the server from the corresponding key store:

- $E_{\mathrm{MX}}^{\mathrm{c}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, c, sid_{\mathrm{c}}, \mathsf{Request}, p_{\mathrm{c}}, pw, 1^{n_{\mathrm{c}}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{LC}}\colon (c, (\mathrm{C}, s, r), \mathsf{GetTime})$
- $\mathcal{F}_{\mathrm{LC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (c, (\mathrm{C}, s, r), \mathsf{Time}, t)$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{ENC}}\colon ((s, c, r), \mathsf{KeyGen})$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon ((s, c, r), \mathsf{KeyGen}, ptr)$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{KS^{ae}}}\colon (s, (\mathrm{C}, c, r), \mathsf{GetKey})$
- $\mathcal{F}_{\mathrm{KS^{ae}}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, (\mathrm{C}, c, r), \mathsf{PublicKey}, pk^{\mathrm{ae}})$

Next, the client encrypts the key k (using the pointer to that key) under pk_s^{ae}, and then uses the pointer to encrypt the request payload under key k:

- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{ENC}}\colon (s, (\mathrm{C}, c, r), \mathsf{Initialize})$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, (\mathrm{C}, c, r), \mathsf{Completed})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{ENC}}\colon (s, (\mathrm{C}, c, r), \mathsf{Enc}, pk^{\mathrm{ae}}, (\mathsf{Key}, ptr))$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, (\mathrm{C}, c, r), \mathsf{Ciphertext}, \varrho_k)$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{ENC}}\colon ((s, c, r), \mathsf{Enc}, ptr, p_{\mathrm{c}})$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon ((s, c, r), \mathsf{Ciphertext}, \varrho_{\mathrm{c}})$

The client constructs the request message $m_{\mathrm{c}} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon r, \mathsf{Time}\colon t, \mathsf{Key}\colon \varrho_k, \mathsf{Body}\colon \varrho_{\mathrm{c}})$, including the encrypted key ϱ_k and the encrypted plaintext ϱ_{c}, and signs and sends the request message:

- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{KS^{sig}}}\colon (c, (\mathrm{C}, s, r), \mathsf{GetKey})$
- $\mathcal{F}_{\mathrm{KS^{sig}}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (c, (\mathrm{C}, s, r), \mathsf{PublicKey}, pk_{\mathrm{c}}^{\mathrm{sig}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{SIG}}\colon (c, (\mathrm{C}, s, r), \mathsf{Sign}, m_{\mathrm{c}})$
- $\mathcal{F}_{\mathrm{SIG}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (c, (\mathrm{C}, s, r), \mathsf{Signature}, \sigma_{\mathrm{c}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to A_{\mathrm{C}}\colon (m_{\mathrm{c}}, \sigma_{\mathrm{c}})$

Upon receival of a potential response message $m_{\mathrm{s}} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon \varrho_{\mathrm{s}})$ in step (B.26), in addition to the above for SA2ME-1, the client decrypts the response payload using its pointer to k:

- $A_{\mathrm{C}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (m_{\mathrm{s}}, \sigma_{\mathrm{s}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{KS^{sig}}}\colon (s, (\mathrm{S}, c, r), \mathsf{GetKey})$
- $\mathcal{F}_{\mathrm{KS^{sig}}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, (\mathrm{S}, c, r), \mathsf{PublicKey}, pk_{\mathrm{s}}^{\mathrm{sig}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{SIG}}\colon (s, (\mathrm{S}, c, r), \mathrm{C}, \mathsf{Init})$
- $\mathcal{F}_{\mathrm{SIG}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}\colon (s, (\mathrm{S}, c, r), \mathrm{C}, \mathsf{Inited})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}} \to \mathcal{F}_{\mathrm{SIG}}\colon (s, (\mathrm{S}, c, r), \mathsf{Verify}, m_{\mathrm{s}}, \sigma_{\mathrm{s}}, pk_{\mathrm{s}}^{\mathrm{sig}})$

- $\mathcal{F}_{\text{SIG}} \to \mathcal{P}_{\text{C}}^{\text{CSA}}$: $(s, (\mathsf{S}, c, r), \mathsf{Verified}, \texttt{true})$
- $\mathcal{P}_{\text{C}}^{\text{CSA}} \to \mathcal{F}_{\text{ENC}}$: $((s, c, r), \mathsf{Dec}, ptr, \varrho_{\text{s}})$
- $\mathcal{F}_{\text{ENC}} \to \mathcal{P}_{\text{C}}^{\text{CSA}}$: $((s, c, r), \mathsf{Plaintext}, p_{\text{s}})$
- $\mathcal{P}_{\text{C}}^{\text{CSA}} \to E_{\text{MX}}^{\text{c}}$: $(s, c, sid_{\text{c}}, \mathsf{Response}, p_{\text{s}})$

If the environment questions whether the client is corrupted, step (B.28), it answers `true` if 1. its own signature scheme, 2. the verifier that the server uses to verify the validity of this client's signature, 3. the server's encryption functionality for this session, or 4. the symmetric key generated for this session are corrupted. Resources that the environment provides for corruption are passed on to the signature scheme, see step (B.27).

4.3.3.2. The Server Functionality $\mathcal{P}_{\text{S}}^{\text{CSA}}$

The server functionality, presented in Appendix B.2.5, is mainly the same as $\mathcal{P}_{\text{S}}^{\text{SA}}$. But in addition, before accepting the message, the server has to decrypt the ciphertext ϱ_k to obtain the key k (step (B.53)) and use the resulting pointer *ptr* to decrypt the ciphertext ϱ_{c} and obtain the request plaintext p_{c} in step (B.54). The server also stores the pointer *ptr* in the list L.

- $\mathcal{P}_{\text{S}}^{\text{CSA}} \to \mathcal{F}_{\text{ENC}}$: $(s, \mathsf{Dec}, \varrho_k)$
- $\mathcal{F}_{\text{ENC}} \to \mathcal{P}_{\text{S}}^{\text{CSA}}$: $(s, \mathsf{Plaintext}, (\mathsf{Key}, ptr))$
- $\mathcal{P}_{\text{S}}^{\text{CSA}} \to \mathcal{F}_{\text{ENC}}$: $((s, c, r), \mathsf{Dec}, ptr, \varrho_{\text{c}})$
- $\mathcal{F}_{\text{ENC}} \to \mathcal{P}_{\text{S}}^{\text{CSA}}$: $((s, c, r), \mathsf{Plaintext}, p_{\text{c}})$

When asked by the environment to respond, the server encrypts the response payload p_{s} using the pointer to the key stored in L, see step (B.55):

- $\mathcal{P}_{\text{S}}^{\text{CSA}} \to \mathcal{F}_{\text{ENC}}$: $((s, c, r), \mathsf{Enc}, ptr, p_{\text{s}})$
- $\mathcal{F}_{\text{ENC}} \to \mathcal{P}_{\text{S}}^{\text{CSA}}$: $((s, c, r), \mathsf{Ciphertext}, \varrho_{\text{s}})$

Similar to the client, upon being asked for a session's corruption status, the server answers `true` if 1. its own signature scheme used in that session, 2. the verifier that the client uses in this session to verify the server's signature, 3. the server's encryption functionality for this session, or 4. the symmetric key generated for this session are corrupted. Resources that the environment provides for corruption are passed on to the signature scheme used in that session (step (B.60)).

4.3.4. Password-Authenticated Two-Round Message Exchange

The system of IITM's implementing $\mathcal{F}_{\text{S2ME}}(leak_{\text{full}}, \texttt{true})$ is defined for any leakage algorithm *leak* and any polynomials p_{sig}, p_{st}, p_{lt}, and p_{ae} by

$$\begin{aligned}\mathcal{P}_{\text{S2ME}}^{\text{PA}}(leak, p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) = \\ !\mathcal{P}_{\text{C}}^{\text{PA}} \mid !\mathcal{P}_{\text{S}}^{\text{PA}} \mid \underline{\underline{!\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid \underline{\underline{!\mathcal{F}_{\text{KS}^{\text{ae}}}}} \mid \underline{\underline{!\mathcal{P}_{\text{SI}}(except_{\text{PA2ME}-1})}} \mid \underline{\underline{!\mathcal{F}_{\text{LC}}}} \mid \\ \underline{\underline{!\mathcal{F}_{\text{SIG}}(p_{\text{sig}})}} \mid \mathcal{F}_{\text{ENC}}(leak, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) \mid \mathcal{F}_{\text{RO}} \ . \quad (4.12)\end{aligned}$$

As this implementation, again, is similar to the ones in Section 4.3.2 and 4.3.3, we mainly point out the differences below.

4.3.4.1. The Client Functionality $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}$

The client functionality, see Appendix B.2.3, does not use digital signatures to sign the request, but instead, see step (B.29), generates a secret message id r, hashes it to obtain H_r, hashes the request $m_{\mathrm{c}} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon H_r, \mathsf{Time}\colon t, \mathsf{Body}\colon p_{\mathrm{c}})$ to obtain the value $H_{m_{\mathrm{c}}}$, and then encrypts a token $m'_c = (\mathsf{SecMsgID}\colon r, \mathsf{Pass}\colon pw, \mathsf{MsgHash}\colon H_{m_{\mathrm{c}}})$ containing the password pw:

- $E^{\mathrm{c}}_{\mathrm{MX}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (s, c, sid_{\mathrm{c}}, \mathsf{Request}, p_{\mathrm{c}}, pw, 1^{n_{\mathrm{c}}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{RO}}\colon (\mathsf{GetRO}, r)$
- $\mathcal{F}_{\mathrm{RO}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (\mathsf{RO}, H_r)$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{LC}}\colon (c, (\mathrm{C}, s, H_r), \mathsf{GetTime})$
- $\mathcal{F}_{\mathrm{LC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (c, (\mathrm{C}, s, H_r), \mathsf{Time}, t)$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{RO}}\colon (\mathsf{GetRO}, m_{\mathrm{c}})$
- $\mathcal{F}_{\mathrm{RO}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (\mathsf{RO}, H_{m_{\mathrm{c}}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{KS^{ae}}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{GetKey})$
- $\mathcal{F}_{\mathrm{KS^{ae}}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{PublicKey}, pk^{\mathrm{ae}}_{\mathrm{s}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{ENC}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{Initialize})$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{Completed})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{ENC}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{Enc}, pk^{\mathrm{ae}}_{\mathrm{s}}, m'_{\mathrm{c}})$
- $\mathcal{F}_{\mathrm{ENC}} \to \mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}\colon (s, (\mathrm{C}, c, H_r), \mathsf{Ciphertext}, \varrho_{\mathrm{c}})$
- $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}} \to A_{\mathrm{C}}\colon (m_{\mathrm{c}}, \varrho_{\mathrm{c}})$

A response is accepted in step (B.30) if it references the hash value of r. As above for the case of SA2ME-1, the client checks the server's signature; thus, the messages are the same as in Section 4.3.2.1.

If the environment questions whether the client is corrupted (step (B.32)), it answers `true` if the server's encryption functionality for this session is corrupted. Resources that the environment provides for corruption are not passed on, because the client does not use a signature scheme, see step (B.31).

4.3.4.2. The Server Functionality $\mathcal{P}_{\mathrm{S}}^{\mathrm{PA}}$

The server functionality is modified accordingly: Upon receival of a message $(m_{\mathrm{c}}, \varrho_{\mathrm{c}})$ with $m_{\mathrm{c}} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{MsgID}\colon H_r, \mathsf{Time}\colon t_{\mathrm{c}}, \mathsf{Body}\colon p_{\mathrm{c}})$, it decrypts the token ϱ_{c} to obtain $m'_{\mathrm{c}} = (\mathsf{SecMsgID}\colon r, \mathsf{Pass}\colon pw, \mathsf{MsgHash}\colon H_{m_{\mathrm{c}}})$ in step (B.65). Differently to the checks performed in SA2ME-1, it does not check a signature (as there is none), but it computes the hash values of r and the message, and it checks if they match H_r and $H_{m_{\mathrm{c}}}$, respectively, and if the password matches the password database of the server:

- $A_{\mathrm{S}} \to \mathcal{P}_{\mathrm{S}}^{\mathrm{PA}}\colon (m_{\mathrm{c}}, \varrho_{\mathrm{c}})$
- $\mathcal{P}_{\mathrm{S}}^{\mathrm{PA}} \to \mathcal{F}_{\mathrm{LC}}\colon (s, \mathrm{S}, \mathsf{GetTime})$

- $\mathcal{F}_{\text{LC}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (s, \mathsf{S}, \mathsf{Time}, t_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{ENC}}\colon (s, \mathsf{Dec}, \varrho_{\text{c}})$
- $\mathcal{F}_{\text{ENC}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (s, \mathsf{Plaintext}, m'_{\text{c}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{RO}}\colon (\mathsf{GetRO}, r)$
- $\mathcal{F}_{\text{RO}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (\mathsf{RO}, H'_r)$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{RO}}\colon (\mathsf{GetRO}, m_{\text{c}})$
- $\mathcal{F}_{\text{RO}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (\mathsf{RO}, H'_{m_{\text{c}}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to E_{\text{MX}}^{\text{s}}\colon (s, c, sid_s, \mathsf{Request}, p_{\text{c}})$

When a response is sent in step (B.67), the server uses an instance of the signature functionality identified by the hash value H_r of r, and the server includes the hash value of r in its response $m_{\text{s}} = (\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon H_r, \mathsf{Body}\colon p_{\text{s}})$, allowing the client to relate the response to its request.

- $E_{\text{MX}}^{\text{s}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (s, c, sid_{\text{s}}, \mathsf{Response}, p_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{RO}}\colon (\mathsf{GetRO}, r)$
- $\mathcal{F}_{\text{RO}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (\mathsf{RO}, H_r)$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{KS}^{\text{sig}}}\colon (s, (\mathsf{S}, c, H_r), \mathsf{GetKey})$
- $\mathcal{F}_{\text{KS}^{\text{sig}}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (s, (\mathsf{S}, c, H_r), \mathsf{PublicKey}, pk_{\text{s}}^{\text{sig}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to \mathcal{F}_{\text{SIG}}\colon (s, (\mathsf{S}, c, H_r), \mathsf{Sign}, m_{\text{s}})$
- $\mathcal{F}_{\text{SIG}} \to \mathcal{P}_{\text{S}}^{\text{PA}}\colon (s, (\mathsf{S}, c, H_r), \mathsf{Signature}, \sigma_{\text{s}})$
- $\mathcal{P}_{\text{S}}^{\text{PA}} \to A_{\text{S}}\colon (m_{\text{s}}, \sigma_{\text{s}})$

Again, if the server is asked for a session's corruption status, it answers `true` if either its own signature scheme used in that session or the verifier that the client uses in this session or the server's encryption functionality are corrupted. Resources that the environment provides for corruption are passed on, see step (B.72).

4.4. Results and Proofs

Our theorem states that the three protocols defined above securely realize the three different parameterizations of the ideal functionality $\mathcal{F}_{\text{S2ME}}$:

Theorem 4.1. *For any polynomials $p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}$, and p_{ae}, and any leakage algorithm $leak_{enc}$ that leaks exactly the length of a message, we have*

$$\mathcal{P}_{\text{S2ME}}^{\text{SA}}(p_{\text{sig}}) \leq^{\text{BB}} \mathcal{F}_{\text{S2ME}}(leak_{\text{full}}, \mathtt{false}) \tag{4.13}$$

$$\mathcal{P}_{\text{S2ME}}^{\text{CSA}}(leak_{enc}, p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) \leq^{\text{BB}} \mathcal{F}_{\text{S2ME}}(leak_{\text{length}}, \mathtt{false}) \tag{4.14}$$

$$\mathcal{P}_{\text{S2ME}}^{\text{PA}}(leak_{enc}, p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}}) \leq^{\text{BB}} \mathcal{F}_{\text{S2ME}}(leak_{\text{full}}, \mathtt{true}) \tag{4.15}$$

Note that in Section 4.5, we show a corollary of this theorem that states that the functionalities on the left-hand sides of the above statements are further securely realizable, using, among others, secure signature and encryption schemes.

In the rest of the section, we prove Theorem 4.1. A full formal proof could, e. g., establish a *bisimulation* (see [Par81] or the proof in Section 5.1.3) between the system consisting of the real protocol and that consisting of the ideal protocol and a simulator. We, instead, define a simulator for each of the statements in the theorem and then argue why the key points in a correctness proof of the bisimulation can be carried out.

But first, we briefly argue why our functionalities fulfill the necessary restrictions on resources defined in [Kü06b, pages 8 et seq.].

Firstly, all versions of the systems $\mathcal{F}_{\text{S2ME}}$ and $\mathcal{P}_{\text{S2ME}}$ are *well-formed* as defined in Section 4.1.1, i. e., the graph induced by enriching tapes between IITM's is acyclic, see Figures 4.3 and 4.4.

Secondly, the restrictions on the running time and the length of messages written on output tapes are fulfilled by all our IITM's: For each message m that is accepted on a non-enriching tape, 1. we either change the state of the IITM to ensure that the same message is not accepted multiple times, or, if a step is carried out multiple times (for example, step (B.18)), the functionality uses a variable like n in $\mathcal{F}_{\text{SM}}$ that keeps track of the amount of resources they received from the environment and decreases n (or sets it to zero) when m is accepted; 2. if m has variable length and may provoke an output message depending on the length of the input, we check if enough resources are available (for example, see step (B.17)), and stop if not.

4.4.1. Signature-Authenticated Two-Round Message Exchange

Proof of (4.13) *in Theorem 4.1.* For the rest of the proof, we fix a polynomial p_{sig} and we write $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ instead of $\mathcal{P}^{\text{SA}}_{\text{S2ME}}(p_{\text{sig}})$ and $\mathcal{F}^{\text{SA}}_{\text{S2ME}}$ instead of $\mathcal{F}_{\text{S2ME}}(\mathit{leak}_{\text{full}}, \texttt{false})$.

To prove (4.13), we construct a simulator $\mathcal{S}^{\text{SA}}_{\text{S2ME}}$ (given in Appendix B.3.1) such that the systems $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S}^{\text{SA}}_{\text{S2ME}} \mid \mathcal{F}^{\text{SA}}_{\text{S2ME}}$ and $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}^{\text{SA}}_{\text{S2ME}}$ are computationally indistinguishable for every adversary $\mathcal{A}$ and every environment $\mathcal{E}$.

The main idea of the simulator is that while interacting with $\mathcal{E}$, $\mathcal{A}$, and all machines that are active in the ideal functionality $\mathcal{F}^{\text{SA}}_{\text{S2ME}}$, it simulates every machine that would be present in a run of the system $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ in such a way that the environment receives the exact same messages on the I/O interface from the machines in $\mathcal{F}^{\text{SA}}_{\text{S2ME}}$ as it would receive from the machines in $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$, and analogously presents network traffic to $\mathcal{A}$ that is indistinguishable to the traffic a real instance of $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ would generate on the same inputs.

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

To show that this simulation indeed works as intended, we argue that for every sequence of messages and instructions sent by $\mathcal{A}$ or $\mathcal{E}$, the simulation is correct in the following sense:

After a message from the adversary or the environment has been processed and control is given back to the adversary or the environment, the state of each machine of $\mathcal{P}^{\mathrm{SA}}_{\mathrm{S2ME}}$ is identical in the simulation (called the "ideal world" in the proof) and in a hypothetical execution of the real protocol (with the same inputs and same random coin tosses, called the "real world").

First, we look at the simulator's variables: It keeps a list in the variable *sessions*, where, for each running protocol session, the server and client identities, the session id sid_A used by the ideal functionality, and the nonce chosen by the simulator for that session are stored. The functions `nonce` and `sid` are used to easily access tuples from the session list. The simulator also keeps an array *status*, which stores the status of simulated instances of $\mathcal{P}^{\mathrm{SA}}_{\mathrm{C}}$ and $\mathcal{P}^{\mathrm{SA}}_{\mathrm{S}}$.

Now, observe that the following invariants hold after a message from the adversary or the environment is processed and control is given back to the adversary or the environment:

(I) For each instance of $\mathcal{P}^{\mathrm{SA}}_{\mathrm{C}}$ running in the real world identified by (s, c, r), there is a simulated session in the simulator with an entry (s, c, sid_A, r) in the list *sessions*, and an instance of $\mathcal{F}_{\mathrm{MX}}$ running in the ideal world which is identified by (s, c, sid_A); and vice-versa.

(II) For each instance of $\mathcal{P}^{\mathrm{SA}}_{\mathrm{S}}$ running in the real world identified by s, there is a simulated server in the simulator with $state[s] \neq \bot$, and an instance of $\mathcal{F}_{\mathrm{SM}}$ is running in the ideal world which is identified by s; and vice-versa

(III) For each tuple (t, r, c, sid_s) with $sid_s \neq \varepsilon$ in the list L of an instance of $\mathcal{P}^{\mathrm{SA}}_{\mathrm{S}}$ in the real world, there is a corresponding instance of $\mathcal{F}_{\mathrm{MX}}$ in the ideal world with $state = 3$, in particular, the instance of $\mathcal{F}_{\mathrm{MX}}$ is not expired; and vice-versa

(IV) With overwhelming probability, the following holds for all tuples (s, c, sid_A, r) in the list *sessions*:

- The instance of the functionality $\mathcal{F}_{\mathrm{MX}}$ identified by (s, c, sid_A) is corrupted on the *client* side if and only if the signature functionality that is identified by $(c, (\mathrm{C}, s, r))$ or the verification functionality identified by $(c, (\mathrm{C}, s, r), \mathrm{S})$ is corrupted.
- The instance of the functionality $\mathcal{F}_{\mathrm{MX}}$ identified by (s, c, sid_A) is corrupted on the *server* side if and only if the signature functionality that is identified by $(s, (\mathrm{S}, c, r))$ or the verification functionality identified by $(s, (\mathrm{S}, c, r), \mathrm{C})$ is corrupted.

This invariant may be broken in a negligible number of cases as explained below in the paragraph about the signature functionality.

We now argue separately for each machine in $\mathcal{P}^{\mathrm{SA}}_{\mathrm{S2ME}}$ that, after a message from the adversary or the environment has been processed and control is given back to the adversary or the environment, the state of the ideal world systems (as far as observable

by the environment and the adversary) are the same as in a hypothetical execution of the real protocol (with the same inputs and same random coin tosses). To that end, we list the messages or instructions that the machines may receive, and argue why the consequences in both the ideal and the real world are the same.

We note that the simulator and the machines in $\mathcal{F}^{SA}_{S2ME}$ do not accept any other messages from the adversary or the environment than the machines in $\mathcal{P}^{SA}_{S2ME}$, hence, the following reasoning suffices.

The Client $\mathcal{P}^{SA}_{C}$.

(B.21) In the real functionality, the receival of a message matching this step starts a new instance of the client functionality $\mathcal{P}^{SA}_{C}$, which wraps, signs, and sends the request. In the ideal functionality, this starts an instance of $\mathcal{F}_{MX}$ (cf. invariant (I)), which forwards its wish to send a request to the simulator (steps (B.1) and (B.91)), which then exactly mimics the steps of $\mathcal{P}^{SA}_{C}$ in `processRequestRequest`.

(B.22) In the real functionality, when a response message is received from the adversary, it is checked and, if the checks are successful, the payload is sent to the environment. The simulator performs exactly the same checks in step (B.94) and `processResponseApproval`, and if successful, forwards the payload to the corresponding instance of $\mathcal{F}_{MX}$, see invariant (I). Now, depending on its corruption status on the server side, $\mathcal{F}_{MX}$ passes on the payload sent by the simulator *or* the payload stored in its internal state, see step (B.5).

Hence, using invariant (IV), we distinguish between corrupted and uncorrupted signature and verification functionalities.

If either one of these functionalities is corrupted, we do not know if the signature is indeed valid. But then we know that the corresponding instance of $\mathcal{F}_{MX}$ is also corrupted on the server side. Thus, it accepts the payload sent by the simulator and passes it on to the environment (even if the request has not been delivered yet, see variable $replied_c$ in step (B.5)).

So we can assume that neither the signature nor the verification functionality is corrupted, but the signature is valid. We now have to inspect $\mathcal{F}_{SIG}$ from [KT08b] to see what guarantees we can draw from that. First observe that we always call the verification functionalities with the correct keys as we use our key store. Then we can conclude that if the signature functionality is not corrupted, but states that the signature is "valid", the message was indeed signed by this functionality (as the message is contained in the set H of $\mathcal{F}_{SIG}$). As we included the nonce r and the constant C in the prefix for the signature scheme, we know that if no collision of message id's occurred (hence, with overwhelming probability), then the signature functionality signed only one response message (note that it may have signed multiple messages through $\mathcal{P}_{SI}$, but none of those could have been a request message through the exception set in $\mathcal{P}_{SI}$). Therefore, we know that $\mathcal{F}_{MX}$ holds exactly the payload in its internal state that the simulator passes on to $\mathcal{F}_{MX}$,

hence, it makes no difference if $\mathcal{F}_{\text{MX}}$ ignores the payload sent by the simulator and just forwards the payload stored internally.

(B.23) Resources received from the environment are passed on to $\mathcal{F}_{\text{KS}^{\text{sig}}}$ in both the real world and the ideal world through steps (B.13) and (B.98).

(B.24) According to invariant (IV), the environment receives the same answer in both the real and the ideal world.

The Server $\mathcal{P}_{\text{S}}^{\text{SA}}$.

(B.33) If the environment starts a new instance of $\mathcal{P}_{\text{S}}^{\text{SA}}$ in the real world, a new instance of $\mathcal{F}_{\text{SM}}$ is started in the ideal world (step (B.14)), which lets the simulator run `processServerInit`, simulating what happens in the real world, and then passing control back to $\mathcal{F}_{\text{SM}}$; also see invariant (II).

(B.34) In the real world, the server simply stores the new resources, while in the ideal world, the message is received from $\mathcal{F}_{\text{SM}}$ in step (B.15) and then passed on to the simulator, who also simply stores it in step (B.96).

(B.35)–(B.39) First, we note that in the real world, the processing of a request is divided into the five steps (B.35) to (B.39), as the adversary may interfere between each of these steps; e. g., when the server wants to retrieve the client's key from the key store in step (B.35), the adversary may block this retrieval.

To not block the whole server, we give the adversary the possibility to just deliver another message m' to the server while the server, e. g., waits in state 2 during the processing of another message m received earlier; but then, we cancel the processing of m and just start to process m'. In the ideal world, this is reflected in that the simulator runs `processRequestApproval` concurrently, but cancels any execution of `processRequestApproval` (and `processResponseRequest`) as soon as a new message arrives.

Then, in `processRequestApproval`, the simulator executes the same steps as the real server would in steps (B.35) to (B.39), with one exception: If a tuple is deleted from the list L in $\mathcal{P}_{\text{S}}^{\text{SA}}$, the simulator explicitly expires the corresponding instances of $\mathcal{F}_{\text{MX}}$, cf. invariant (III).

If the message is approved and the server in the real world would deliver it to the environment, the simulator sends the payload to the corresponding instance of $\mathcal{F}_{\text{MX}}$. Analogously to the reasoning for the client above in step (B.22), it does not matter if the client side of the instance of $\mathcal{F}_{\text{MX}}$ is corrupted, as the correct payload is always delivered to the environment.

(B.40)–(B.42) Here, the situation with the three steps (B.40) to (B.42) is analogous to the five steps discussed above; the simulator summarizes the processing in step (B.93) and in `processResponseRequest`, and the processing may be interrupted (i. e., canceled) by the adversary.

But first, note that due to invariant (III), we know that if the environment responds to a non-existent or an expired session, it receives an error message in both the real and the ideal world: In the real world, the server simply handles all requests from the environment and decides using the list L, while in the ideal world, there are two cases, both handled in step (B.7): An expired instance of $\mathcal{F}_{\mathrm{MX}}$ sends an error message directly, but if the environment tries to call a session that never existed, a new instance of $\mathcal{F}_{\mathrm{MX}}$ is started, sends the error message, and then terminates. For the environment, both variants are not distinguishable.

Again, the simulator executes the same steps as the real server does, but while the real server updates the entry in the list L by overwriting sid_s with ε, the simulator simply updates a boolean value in its copy of L from `false` to `true`. Both in the real and the ideal world, the resulting message is sent to the adversary.

(B.43) If the server is reset in the real world, mainly the list L is emptied. This is reflected in the ideal world by the simulator in that it expires all existing sessions in step (B.95), cf. invariant (III).

(B.45) In the real world, the server passes the resources on to the corresponding signature scheme. In the ideal world, the resources are received from $\mathcal{F}_{\mathrm{MX}}$ in step (B.13) and then sent to the simulator, who passes it on to the simulated signature scheme (B.98).

(B.46) As above for the client, according to invariant (IV), the environment receives the same answer in both the real and the ideal world. But here we note that the environment ist able to question the corruption status of any session, even expired ones. This is reflected in the real world by using separate lists, L and L_{cor}, where deletions and updates only are applied to the first list. In the ideal world, $\mathcal{F}_{\mathrm{MX}}$ even answers in expired sessions, see step (B.12).

The Signature Functionality $\mathcal{F}_{\mathrm{SIG}}(p_{\mathrm{sig}})$. The simulator directly mimics all instances of the ideal signature functionality, thus, its state is the same in both worlds. In addition, through steps (B.99) and (B.100), the simulator ensures that invariant (IV) holds.

Note that the steps (B.99) and (B.100) do not cover all cases—this is the reason why invariant (IV) only holds with overwhelming probability: The adversary may corrupt signature or verification functionalities *before* a corresponding instance of $\mathcal{F}_{\mathrm{MX}}$ is initialized. But as the nonce r identifying the instances that are later connected by the simulator to an instance of $\mathcal{F}_{\mathrm{MX}}$ is chosen randomly from $\{0,1\}^{\eta}$, the probability that the adversary "hits" such an instance is negligible.

The Key Store $\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}$, the Local Clock $\mathcal{F}_{\mathrm{LC}}$, and the Signature Interface $\mathcal{P}_{\mathrm{SI}}$. These three functionalities are simulated by the simulator as-is, i. e., they are exactly the same in both worlds. □

4.4.2. Confidential Signature-Authenticated Two-Round Message Exchange

Proof of (4.14) *in Theorem 4.1.* For the rest of the proof, fix a leakage algorithm *leak* that leaks exactly the length of the message and polynomials p_{sig}, p_{st}, p_{lt}, and p_{ae}. As above, we write $\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$ instead of $\mathcal{P}^{\text{CSA}}_{\text{S2ME}}(leak, p_{\text{sig}}, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}})$ and we use $\mathcal{F}^{\text{CSA}}_{\text{S2ME}}$ instead of $\mathcal{F}_{\text{S2ME}}(leak_{\text{length}}, \texttt{false})$ for the rest of the proof.

To prove (4.14), we again construct a simulator $\mathcal{S}^{\text{CSA}}_{\text{S2ME}}$, see Appendix B.3.2, such that the systems $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S}^{\text{CSA}}_{\text{S2ME}} \mid \mathcal{F}^{\text{CSA}}_{\text{S2ME}}$ and $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}^{\text{CSA}}_{\text{S2ME}}$ are computationally indistinguishable for every adversary $\mathcal{A}$ and every environment $\mathcal{E}$.

The simulator $\mathcal{S}^{\text{CSA}}_{\text{S2ME}}$ mainly works as $\mathcal{S}^{\text{SA}}_{\text{S2ME}}$ above. An additional key point in this proof is the "transitivity" of the leakage algorithms involved: In the ideal world, our simulator usually only learns the leakage (under $leak_{\text{length}}$) of the payloads in messages. But as the algorithm *leak* leaks exactly the length of the message, it does not matter if we provide that leakage algorithm with an original payload or with that payload's leakage unter $leak_{\text{length}}$.

Again, we argue that for every sequence of messages and instructions sent by $\mathcal{A}$ or $\mathcal{E}$, the simulation is correct in the following sense:

After a message from the adversary or the environment has been processed and control is given back to the adversary or the environment, the state of each machine of $\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$ is identical in the simulation and in a hypothetical execution of the real protocol, except for the (randomly chosen) leakage.

Analogously to the invariants (I) to (IV), the following invariants hold after a message from the adversary or the environment is processed and control is given back to the adversary or the environment:

(V) For each instance of $\mathcal{P}^{\text{CSA}}_{\text{C}}$ running in the real world identified by (s, c, r), there is a simulated session in the simulator with an entry (s, c, sid_A, r) in the list *sessions*, and an instance of $\mathcal{F}_{\text{MX}}$ running in the ideal world identified by (s, c, sid_A); and vice-versa

(VI) For each instance of $\mathcal{P}^{\text{CSA}}_{\text{S}}$ running in the real world identified by s, there is a simulated server in the simulator with $state[s] \neq \perp$, and an instance of $\mathcal{F}_{\text{SM}}$ is running in the ideal world which is identified by s; and vice-versa

(VII) For each tuple (t, r, c, sid_s, ptr) with $sid_s \neq \varepsilon$ in the list L of an instance of $\mathcal{P}^{\text{CSA}}_{\text{S}}$ in the real world, there is a corresponding instance of $\mathcal{F}_{\text{MX}}$ in the ideal world with $state = 3$, in particular, the instance of $\mathcal{F}_{\text{MX}}$ is not expired; and vice-versa

(VIII) With overwhelming probability, the following holds for all tuples (s, c, sid_A, r) in the list *sessions*:

- The instance of the functionality $\mathcal{F}_{\text{MX}}$ identified by (s, c, sid_A) is corrupted on the *client* side if and only if at least one of the following four is corrupted:
  a) the signature functionality identified by $(c, (\text{C}, s, r))$,

b) the verification functionality identified by $(c, (\mathsf{C}, s, r), \mathsf{S})$,

c) the encryption functionality identified by $(s, (\mathsf{C}, c, r))$, or

d) the symmetric key to which a pointer is stored in $ptrs[s, c, r]$ by the simulator.

- The instance of the functionality $\mathcal{F}_{\text{MX}}$ identified by (s, c, sid_A) is corrupted on the *server* side if and only if at least one of the following four is corrupted:

  a) the signature functionality identified by $(s, (\mathsf{S}, c, r))$,

  b) the verification functionality identified by $(s, (\mathsf{S}, c, r), \mathsf{C})$,

  c) the encryption functionality identified by $(s, (\mathsf{C}, c, r))$, or

  d) the symmetric key to which a pointer is stored in $ptrs[s, c, r]$ by the simulator.

Again, we argue for the machines in $\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$ that, after a message from the adversary or the environment has been processed and control is given back to the adversary or the environment, the observable state of the ideal world systems is the same as in a hypothetical execution of the real protocol (with the same inputs and same random coin tosses).

This is enough for the proof as the simulator and the machines in $\mathcal{F}^{\text{CSA}}_{\text{S2ME}}$ only accept messages from the adversary or the environment that are also accepted by machines in $\mathcal{P}^{\text{CSA}}_{\text{S2ME}}$.

The Client $\mathcal{P}^{\text{CSA}}_{\text{C}}$.

(B.25) As above, as soon as the environments starts a client instance in the real world, it starts an instance of $\mathcal{F}_{\text{MX}}$ in the ideal world, which in turn leads the simulator to start a simulated instance of $\mathcal{P}^{\text{CSA}}_{\text{C}}$, see invariant (V). The instructions in step (B.25) are mimicked by the simulator in `processRequestRequest`, including the additional steps for keeping the payload confidential:

In both worlds, the client instructs the encryption functionality to generate a fresh symmetric key and return a pointer to that key. Next, the client encrypts the symmetric key using asymmetric encryption by sending the tuple (Key, ptr) to the encryption functionality. Then, the client sends the request payload and the key pointer to the encryption functionality, which encrypts the payload using symmetric encryption. The client includes both ciphertexts in the request message.

If both symmetric and asymmetric encryption are uncorrupted, the messages sent to the adversary are generated differently in both worlds, so we have to argue why the messages sent to the adversary are indistinguishable.

But first we note that if either the symmetric key generated for encrypting the payload is corrupted (which the adversary can only decide upon generation, not at a later time) or the public key of the server is corrupted (which also is modeled as

static corruption, i. e., the adversary can only corrupt the key at generation time), the simulator retrieves the original payload from the client (using step (B.11)).

Now, assume that both encryption schemes are uncorrupted. We first take a look at the real world. Here, the client sends the original payload p_c to the encryption functionality, which then uses the leakage algorithm *leak* to determine a bit string that is encrypted using the algorithm provided by the adversary. In the ideal world, the functionality $\mathcal{F}_{\text{MX}}$ applies $leak_{\text{length}}$ to p_c, sends the resulting bit string p'_c to the simulator, which passes it on to the encryption functionality where *leak* is applied to p'_c.

Thus, we know that the leakage algorithm *leak* in both worlds is applied to bit strings of the same length. But then, using the assumption that *leak* leaks exactly the length of the message, we know that the outputs are indistinguishable.

(B.26) Both in the real and in the ideal world, the (simulated) client checks any incoming response message, decrypts the contained payload, and forwards the plaintext to the environment if no error occurred.

In more detail, the real client (in step (B.26)) as well as the simulator (in subroutine `processResponseApproval`) first perform the same checks as in SA2ME. After those checks, the plaintext is decrypted using the key generated for the request message. At this point, we again have to argue why the same (correct) plaintext is received by the environment in both worlds.

Invariant (VIII) allows us to distinguish between corrupted and uncorrupted signature, verification, and encryption functionalities.

If one of those is corrupted, we do neither know if the signature is valid nor do we know if the ciphertext was encrypted by that encryption scheme. But again, in this case, the corresponding instance of $\mathcal{F}_{\text{MX}}$ is corrupted on the server side, so $\mathcal{F}_{\text{MX}}$ accepts any payload delivered by the simulator in step (B.5)—thus, the exact same payload is computed and delivered in both worlds (without any guarantee which bit string that is).

But when we assume that neither signature nor verification nor encryption functionalities are corrupted, then the same arguments as above for SA2ME hold, and we know that the message is the server's valid response sent by the server in step (B.57) (in the real world) or the simulator in `processResponseRequest` (in the ideal world, respectively).

(B.27) The received resources are passed on to $\mathcal{F}_{\text{KS}^{\text{sig}}}$ in both worlds. Note that, in contrast to the signature functionality, the encryption functionality does not require resources for corrupted operations.

(B.28) See invariant (VIII).

The Server $\mathcal{P}_S^{CSA}$.

(B.47) Starting new servers is similar to the implementation $\mathcal{P}_S^{SA}$ (cf. invariant (VI)), with the only difference being that both the real server and the simulator initialize the server's key, which gives the adversary the only opportunity to corrupt that key.

(B.48) Again, resources from the environment are treated equally in both worlds, see steps (B.15) and (B.107).

(B.49)–(B.54) In `processRequestApproval`, the simulator mimics what the real server does in steps (B.49) to (B.54): fetching the public key needed for verifying the signature from the key server, getting the current time, verifying the client's signature, decrypting the symmetric session key (using the server's public key for the asymmetric encryption scheme) as well as the payload (using the symmetric session key that was just decrypted), and finally executing the what the protocol specifies on the server side. Again, if tuples are deleted from L in $\mathcal{P}_S^{CSA}$, the simulator explicitly expires the corresponding instances of $\mathcal{F}_{MX}$, cf. invariant (VII).

As above, in case of corruption, nothing is guaranteed for the plaintext, e. g., the decryption might have failed (thus, $p_c = \perp$ is sent to the environment). But again, this is consistent in both worlds, cf. invariant (VIII). If no corruption occurred that is relevant to this session, a correct signature—as above—allows us to conclude that the message indeed originated from the simulator, and hence, the corresponding instance of $\mathcal{F}_{MX}$ holds the correct plaintext, which is delivered to the environment in both worlds.

(B.55)–(B.57) In `processResponseRequest`, the simulator mimics what the real server does in steps (B.55) to (B.57): checking if sending a response is still permitted, encrypting the payload using the symmetric session key stored in L, and signing the response.

As for the request message, the simulator only receives the leakage of the response payload from the ideal functionality. Again (see our reasoning for step (B.25) above), we know that it makes no difference if the simulator sends the received leakage to $\mathcal{F}_{ENC}$ or if $\mathcal{P}_S^{CSA}$ sends the real payload to $\mathcal{F}_{ENC}$, since $\mathcal{F}_{ENC}$ chooses a bit string uniformly at random from the same distribution in both cases that is then encrypted in $\mathcal{F}_{ENC}$.

(B.58) If the server is reset in the real world, mainly the list L is emptied. This is reflected in the ideal world by the simulator in that it expires all existing sessions in step (B.106), cf. invariant (VII).

(B.60) Again, the server passes on the resources to the corresponding signature scheme in both worlds.

(B.61) Again, according to invariant (VIII), the environment receives the same answer in both the real and the ideal world.

The Signature Functionality $\mathcal{F}_{\mathrm{SIG}}(p_{\mathrm{sig}})$. Again, the ideal signature functionality is directly simulated, and the simulator, through steps (B.111) and (B.112), ensures that invariant (VIII) holds.

The Encryption Functionality $\mathcal{F}_{\mathrm{ENC}}(\mathit{leak}, p_{\mathrm{st}}, p_{\mathrm{lt}}, p_{\mathrm{ae}})$. Similar to the above, the simulator directly simulates the ideal encryption functionality and uses step (B.110) to ensure that invariant (VIII) holds.

The Key Stores $\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}$ and $\mathcal{F}_{\mathrm{KS}^{\mathrm{ae}}}$, the Local Clock $\mathcal{F}_{\mathrm{LC}}$, and the Signature Interface $\mathcal{P}_{\mathrm{SI}}$. These four functionalities are simulated by the simulator as-is, i. e., they are exactly the same in both worlds. □

4.4.3. Password-Authenticated Two-Round Message Exchange

Proof of (4.15) *in Theorem 4.1.* For the rest of the proof, fix a leakage algorithm *leak* that leaks exactly the length of the message and polynomials p_{sig}, p_{st}, p_{lt}, and p_{ae}. Analogously to above, we write $\mathcal{P}^{\mathrm{PA}}_{\mathrm{S2ME}}$ instead of $\mathcal{P}^{\mathrm{PA}}_{\mathrm{S2ME}}(\mathit{leak}, p_{\mathrm{sig}}, p_{\mathrm{st}}, p_{\mathrm{lt}}, p_{\mathrm{ae}})$ and $\mathcal{F}^{\mathrm{PA}}_{\mathrm{S2ME}}$ instead of $\mathcal{F}_{\mathrm{S2ME}}(\mathit{leak}_{\mathrm{full}}, \mathtt{true})$ for the rest of the proof.

Again, to prove (4.15), we construct a simulator $\mathcal{S}^{\mathrm{PA}}_{\mathrm{S2ME}}$ (given in Appendix B.3.3) such that, for every adversary $\mathcal{A}$ and every environment $\mathcal{E}$, the system $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S}^{\mathrm{PA}}_{\mathrm{S2ME}} \mid \mathcal{F}^{\mathrm{PA}}_{\mathrm{S2ME}}$ is computationally indistinguishable from the system $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P}^{\mathrm{PA}}_{\mathrm{S2ME}}$.

Besides the intended usage of the protocol, the adversary may also start server-only sessions if it knows the password of a client by simply generating a request message. Note that in such sessions, the adversary is able to control the value r; therefore, the adversary can provoke a collision: The adversary starts a server-only session using some value r, then sends enough messages to force the server to delete the entry with r from its list L, and then starts another server-only session using the same value r. Therefore, we cannot assume that no collisions occur for the values of r over the run of the simulator, but we can assume the following:

1. For full sessions, the probability that a client chooses a secret message id r that collides with a secret message id previously chosen by another client *or* the adversary is negligible.

2. Similarly, for full sessions, the collision resistance of the hash function (or the random oracle in our case) implies that the probability that a client chooses a secret message id r such that r's hash value collides with the hash value of a secret message id previously chosen by another client is negligible.

3. As long as the protocol only publishes the ciphertext of r and the hash value of r, the probability that the adversary is able to "guess" the value of any r used in a full session is also negligible (given that the adversary did not, e. g., obtain that value by corrupting that session).

4. From the above, the function $\texttt{sid}^{\neg\texttt{so}}$ of our simulator is able to either unambiguously select an entry in the list *sessions* or return $\perp$ if no matching entry exists.

Similar to the proofs above, some invariants hold after a message from the adversary or the environment is processed and control is given back to the adversary or the environment:

(IX) For each instance of $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}$ running in the real world identified by (s, c, r), there is a simulated session in the simulator with an entry $(s, c, sid_A, r, \texttt{false})$ in the list *sessions*, and an instance of $\mathcal{F}_{\mathrm{MX}}$ running in the ideal world which is identified by (s, c, sid_A); and vice-versa

(X) For each instance of $\mathcal{P}_{\mathrm{S}}^{\mathrm{PA}}$ running in the real world identified by s, there is a simulated server in the simulator with $state[s] \neq \perp$, and an instance of $\mathcal{F}_{\mathrm{SM}}$ is running in the ideal world which is identified by s; and vice-versa

(XI) For each tuple (t, r, c, sid_s) with $sid_s \neq \varepsilon$ in the list L of an instance of $\mathcal{P}_{\mathrm{S}}^{\mathrm{PA}}$ in the real world, there is a corresponding instance of $\mathcal{F}_{\mathrm{MX}}$ in the ideal world with $state = 3$, in particular, the instance of $\mathcal{F}_{\mathrm{MX}}$ is not expired; and vice-versa

(XII) With overwhelming probability, the following conditions hold for all tuples $(s, c, sid_A, r, \textit{server-only})$ in the list *sessions*:
- The instance of the functionality $\mathcal{F}_{\mathrm{MX}}$ identified by (s, c, sid_A) is corrupted on the *client* side if and only if the encryption functionality identified by $(s, (\mathrm{C}, c, r))$ is corrupted.
- The instance of the functionality $\mathcal{F}_{\mathrm{MX}}$ identified by (s, c, sid_A) is corrupted on the *server* side if and only if the encryption functionality identified by $(s, (\mathrm{C}, c, r))$, the signature functionality identified by $(s, (\mathrm{S}, c, H(r)))$ or the verification functionality identified by $(s, (\mathrm{S}, c, H(r)), \mathrm{C})$ is corrupted.

Again, we now inspect each machine in $\mathcal{P}_{\mathrm{S2ME}}^{\mathrm{PA}}$ and argue that, after a message from the adversary or the environment has been processed and control is given back to the adversary or the environment, the state of the ideal world systems (as far as observable by the environment and the adversary) are the same as in a hypothetical execution of the real protocol (with the same inputs and same random coin tosses).

The Client $\mathcal{P}_{\mathrm{C}}^{\mathrm{PA}}$.

(B.29) In general, the simulator receives the request through step (B.1) and then mimics step (B.29) of the real functionality in `processRequestRequest`, ensuring that invariant (IX) holds.

One difference is that the simulator initially does not know the password used by the client, as only the length of the password is sent in step (B.1). If the password is encrypted for a key that is known to the adversary, this would enable the adversary to distinguish both worlds. Thus, we have to distinguish if the public encryption key of the server is known to the adversary, i. e., if the key is corrupted.

As the public key encryption functionality offers only static corruption and as the simulator retrieves the key for public key encryption from the key store (which initializes the key if that has not been done previously), the simulator knows if the key is corrupted after retrieving the key.

Also, if the key is corrupted, then we know that the corresponding instance of $\mathcal{F}_{\text{MX}}$ is also corrupted: If the key was corrupted before the call to the subroutine `processRequestRequest`, then the instance of $\mathcal{F}_{\text{MX}}$ is corrupted as a first step in `processRequestRequest`; otherwise, if the key is corrupted by the adversary during the processing of `processRequestRequest`, step (B.122) ensures that the instance of $\mathcal{F}_{\text{MX}}$ is corrupted because of its entry in *sessions*. Hence, invariant (XII) holds.

Therefore, if the key is corrupted, the corresponding instance of $\mathcal{F}_{\text{MX}}$ reveals the password upon request by the simulator. Now the simulator can use the password which would also be used in the real world. Otherwise, if the key is not corrupted, the simulator just uses a random bitstring of length l_{pw} as the password.

We again use that by the definition of $\mathcal{F}_{\text{ENC}}$ (more precisely, by the definition of $\mathcal{F}_{\text{pke}}$ in [KT09b]), for an uncorrupted key, only the leakage of a message is really encrypted. Thus, for the resulting ciphertext in case of an uncorrupted key, it does not matter if the simulator used the original password or another bitstring of the same length.

(B.30) The case of the response is similar to $\mathcal{P}_{\text{C}}^{\text{SA}}$, as the server signs its response similar to the signature-authenticated version. As usual, the simulator mimics the steps of the client in the real world.

If there is no full session (see below where we discuss the server's steps), the simulator simply drops the message (see `processResponseApproval`). This is equivalent to what happens in the real world, as clients are only running for full sessions.

If, on the other hand, an entry for a full session is fetched from *sessions* by the simulator, we know that in the real world, a client exists that accepts the message in CheckAddress mode, as it performs the same check, namely testing if the Ref value of an incoming message matches H_r.

Now, if a message is accepted by the simulator, it forwards the contained payload to an instance of $\mathcal{F}_{\text{MX}}$. Again (as above for $\mathcal{P}_{\text{C}}^{\text{SA}}$), depending on the corruption status on the server side, the addressed instance of $\mathcal{F}_{\text{MX}}$ passes on the payload sent by the simulator *or* the payload stored in its internal state, see step (B.5).

If the functionality of $\mathcal{F}_{\text{MX}}$ identified by sid_A is corrupted on the server side, by invariant (XII) we do not know if the signature scheme or the encryption scheme used by the server is corrupted, but $\mathcal{F}_{\text{MX}}$ accepts the payload sent by the simulator and passes it on to the environment, just as the real server does.

The interesting part here is the case in which the functionality of $\mathcal{F}_{\text{MX}}$ identified by sid_A is not corrupted on the server side. Then, by invariant (XII) we know that

neither the server's public key for encryption nor the signature or verification scheme identified by H_r are corrupted. But there may be two problems: First, the adversary may know the client's password, and thus could be able to start server-only sessions between this client and the server, and then try to make the client accept one of the server's responses from these sessions. Secondly, the response messages do not directly contain the secret message id r, but only the hash value of r.

So, assume the accepted message contained some Ref value H_r. By the collision resistance of the hash function (or the random oracle, in our case), we know that with overwhelming probability only one full session exists in which a client chose a value r such that the hash value of r equals H_r. From the definition of $\mathcal{F}_{\text{ENC}}$ in combination with the assumption that the server's key is not corrupted we know that the adversary cannot obtain any knowledge from the ciphertext of the request message (in which r is encrypted) as only the length of r is leaked (which the adversary already knows). The preimage resistance of the hash function (or the random oracle) then guarantees that the adversary, from knowing just the value H_r, has only a negligible chance to compute any value r' such that the hash value of r' equals H_r. Thus, the adversary cannot obtain a validly signed response containing H_r using server-only sessions.

Then we can conclude that the signature functionality identified by H_r signed only one message, and as the (uncorrupted) verification functionality already stated that the signature is "valid", the message was indeed signed by the signature functionality and we know that the corresponding instance of $\mathcal{F}_{\text{MX}}$ holds exactly the payload in its internal state that the simulator passes on to $\mathcal{F}_{\text{MX}}$.

(B.31) As the client does not use a signature scheme, in both the real world and the ideal world, resources received are just dropped.

(B.32) According to invariant (XII), the environment receives the same answer in both the real and the ideal world.

The Server $\mathcal{P}_{\text{S}}^{\text{PA}}$.

(B.33) As above for $\mathcal{P}_{\text{S}}^{\text{SA}}$, the simulator is called by $\mathcal{F}_{\text{SM}}$ (see step (B.14)) and runs `processServerInit`, simulating what happens in the real world; see invariant (X).

(B.63) This step is analogous to the corresponding step (B.63) in $\mathcal{P}_{\text{S}}^{\text{SA}}$.

(B.64)–(B.37) If a message is received and processed by the server in the real world in steps (B.64) to (B.66), the simulator runs `processRequestApproval` and executes analogous steps to what the real server would execute.

As above for $\mathcal{P}_{\text{S}}^{\text{SA}}$, a difference is that if a tuple is deleted from the list L in $\mathcal{P}_{\text{S}}^{\text{PA}}$, the simulator explicitly expires the corresponding instances of $\mathcal{F}_{\text{MX}}$, again see invariant (XI).

Another difference is the handling of passwords: In the real world, the server has access to all passwords and can simply test if the user-supplied password is correct. The simulator, on the other hand, does not have access to the password function U of a server, it has to use steps (B.9) and (B.18) to determine if a single password is correct. We have to distinguish two cases, depending on the corruption status of the encryption key of s and the secret message id r contained in the encrypted part ϱ_c of the message:

1. The server's key is not corrupted and the secret message id r belongs to a full session, i. e., $cor[s] = \texttt{false}$ and $\texttt{sid}^{\neg\texttt{so}}(s, c, r) \neq \perp$.

   First, by definition of $\mathcal{F}_{\text{ENC}}$, the adversary is not able to learn the secret message id r encoded in request messages sent to him; and as the value r is chosen randomly, the adversary has a negligible probability to "hit" a valid secret message id r by just guessing. Hence, we know that this encrypted part ϱ_c was created by the simulator in an earlier run of `processRequestRequest`.

   As explained above for step (B.29), the simulator in this case just encoded an arbitrary bit string in `processRequestRequest` instead of the real password. But through invariant (XI) we know there is a corresponding instance of $\mathcal{F}_{\text{MX}}$ containing the password provided by the environment for this session.

   Thus, the simulator can simply ask the corresponding instance of $\mathcal{F}_{\text{MX}}$ to send the password to $\mathcal{F}_{\text{SM}}$ for testing.

2. The server's key is corrupted, or the secret message id r does not belong to a full session, i. e., $cor[s] = \texttt{true}$ or $\texttt{sid}^{\neg\texttt{so}}(s, c, r) = \perp$.

   If the server's key is corrupted, then the simulator encoded the correct password in `processRequestRequest`. If the secret message id r does not belong to a full session, we know that the simulator did not create that encrypted part (note that entries are only removed from L, not from *sessions*). Therefore, in both cases, the simulator just uses the password that is encoded in the request message and sends it to $\mathcal{F}_{\text{SM}}$ for testing.

A third difference between the ideal world and the real world is the explicit modeling of server-only sessions: As the password of a user may be known to the adversary, it may start a session that appears to be originating from a client, but where the request message simply is forged by the adversary.

While in the real world, a server may accept such a message and not even notice that the real client was not involved, our functionality $\mathcal{F}_{\text{MX}}$ has an explicit modeling of server-only sessions, and the simulator knows if a session is initiated by the adversary without a client by simply looking up the secret message id r in the *sessions* list using the function $\texttt{sid}^{\neg\texttt{so}}$. Thus, if a server-only session is started by the adversary and the message is accepted for delivery by the simulator, it sends a message to $\mathcal{F}_{\text{SM}}$ which starts a new instance of $\mathcal{F}_{\text{MX}}$.

Now, for full sessions, we again have to argue why the correct payload is delivered to the environment by $\mathcal{F}_{\text{MX}}$. In case of a corrupted session of $\mathcal{F}_{\text{MX}}$ on the

client side, this is clear as $\mathcal{F}_{\text{MX}}$ just delivers the payload sent by the simulator. If, however, the session of $\mathcal{F}_{\text{MX}}$ on the client side is not corrupted, then we know that $cor[s] = \texttt{false}$. Thus, we know that the encrypted part ϱ_c was created by `processRequestRequest` as we only consider full sessions. Hence, the hash value H_{m_c} included in ϱ_c is the same as the one calculated when the request was sent. As we also checked that the payload sent by the adversary has a hash value of $H'_{m_c} = H_{m_c}$, by the assumptions that the hash function (or in our case the random oracle) is second-preimage resistant, we know that the payload delivered by the adversary is the same as the one stored in the corresponding session of $\mathcal{F}_{\text{MX}}$.

(B.67)–(B.69) For the response, the server part is similar to $\mathcal{P}_{\text{S}}^{\text{SA}}$: Both in the real and in the ideal world, the server signs the response message (using a signature scheme identified by H_r) and delivers the signed response message to the adversary.

(B.70) Again, a reset in both world results in an empty list L, and again, the simulator empties the list by expiring all sessions in `processServerReset`, also see invariant (XI).

(B.72) This step is analogous to the corresponding step (B.45) in $\mathcal{P}_{\text{S}}^{\text{SA}}$.

(B.73) According to invariant (XII), the environment receives the same answer in both the real and the ideal world.

The Signature Functionality $\mathcal{F}_{\text{SIG}}(p_{\text{sig}})$ and the Encryption Functionality $\mathcal{F}_{\text{ENC}}(leak, p_{\text{st}}, p_{\text{lt}}, p_{\text{ae}})$. These functionalities are simulated, using steps (B.122) through (B.124), to ensure that invariant (XII) holds.

The Key Stores $\mathcal{F}_{\text{KS}^{\text{sig}}}$ and $\mathcal{F}_{\text{KS}^{\text{ae}}}$, the Local Clock $\mathcal{F}_{\text{LC}}$, and the Signature Interface $\mathcal{P}_{\text{SI}}$. These four functionalities are simulated by the simulator as-is, i. e., they are exactly the same in both worlds. □

4.5. Implementing the Protocols

Our realizations $\mathcal{P}_{\text{S2ME}}$ above still contain ideal functionalities that cannot be implemented on a real machine directly. Hence, in this section, we explain how the ideal functionalities occurring in $\mathcal{P}_{\text{S2ME}}$ can be securely realized, with the exception of $\mathcal{F}_{\text{RO}}$ (see notes in Sections 2.1.3.5 and 4.3.1.6).

4.5.1. Uniform and Non-Uniform Adversaries

The security notions for signature and encryption schemes that we use later in this section and that we recalled in Section 2.1.3 are defined with respect to uniform adversaries, i. e., the adversary does only receive the security parameter and no auxiliary input.

In contrast, the IITM framework (as well as [Can04]) defines security with non-uniform environments which may receive additional auxiliary input (and as the environment may cooperate with the adversary, this results in non-uniform adversaries), see Section 4.1.2.

Thus, when bridging the gap between ideal functionalities in the IITM framework and realizations that use signature or encryption schemes secure in a "traditional" sense, one either has to adapt the traditional security notions to the non-uniform case or restrict the IITM environment and adversary to the uniform case.

As explained in [KT08b, KT09b], for our case, both approaches would be possible. Here, we follow the second approach: We use $\leq^{\text{BB-noaux}}$ to denote secure realization as defined above, but for the case that the environment does only receive ε as auxiliary input. See [KT08b] for more details.

4.5.2. Signature Functionality $\mathcal{F}_{\text{SIG}}$

The signature functionality $\mathcal{F}_{\text{SIG}}$ can be implemented using an EUF-CMA secure signature scheme as shown in [KT08b, Theorem 5]. More precisely, the authors of that paper define a straight-forward "wrapper" $\mathcal{P}_{\text{SIG}}$ such that if Ω is an EUF-CMA secure signature scheme that is bounded by some polynomial p_{sig}, and if $\mathcal{T}_{\text{sig}}$ and $\mathcal{T}_{\text{ver}}$ are defined as above, $\mathcal{P}_{\text{SIG}}(\Omega, \mathcal{T}_{\text{sig}}, \mathcal{T}_{\text{ver}}) \leq^{\text{BB-noaux}} \mathcal{F}_{\text{SIG}}(\mathcal{T}_{\text{sig}}, \mathcal{T}_{\text{ver}}, p_{\text{sig}})$.

In our realizations above, $\mathcal{F}_{\text{SIG}}$ occurs in the multi-session multi-user version $\underline{\underline{\mathcal{F}_{\text{SIG}}}}$. Hence, if $\mathcal{F}_{\text{SIG}}$ is implemented by a signature scheme, this would imply that for each user and each session (i. e., for each message that is sent), a new instance of the signature scheme is used.

This is unrealistic and can be avoided by applying a joint-state theorem [KT08b, Theorem 6] allowing different sessions to use the same key: Essentially, a wrapper $\mathcal{P}^{\text{JS}}_{\text{SIG}}$ managing different sessions is used to access the signature functionalities. Thus, instead of one key per party and per session ($!\underline{\underline{\mathcal{F}_{\text{SIG}}}}$), one can use only a single key for each party ($!\underline{\mathcal{F}_{\text{SIG}}}$), as in a realistic public key infrastructure.

Note that this adds unnatural prefixes to messages, see Section 4.6.4 for comments and an alternative approach.

We note that we communicate with signature schemes with prefixes *pid*, *sid* where *pid* is the identity of a party (usually the client's or the server's identity) and *sid* is a session identifier. In contrast, $\mathcal{P}^{\text{JS}}_{\text{SIG}}$ expects the prefixes to messages to appear in the order $sid, pid, \ldots$. Therefore, with $\mathcal{P}^{\text{JS}'}_{\text{SIG}}$ we denote the version of $\mathcal{P}^{\text{JS}}_{\text{SIG}}$ where the order of these two prefixes is inverted.

4.5.3. Signature Interface $\mathcal{P}_{\text{SI}}$

In our modeling, the signature interface $\mathcal{P}_{\text{SI}}$ grants the adversary (limited) access to the keys used in the protocol. Usually, one would not want to realize this functionality

at all, i. e., not allow any outside access to the keys. To that end, one can simply implement $\mathcal{P}_{\text{SI}}$ by a functionality $\mathcal{P}_{\text{SI}}^{\text{dummy}}$ that just accepts the incoming resources from the environment, but does not send out any messages, see Appendix B.2.10.

Note that, however, including the signature interface functionality allowed us to prove the protocol secure *even if* the adversary has access to the keys. For some further notes on this functionality, see Section 4.6.

4.5.4. Encryption Functionality $\mathcal{F}_{\text{ENC}}$

The encryption functionality $\mathcal{F}_{\text{ENC}}$ defined in Section 4.3.1.4 is composed of a symmetric encryption functionality $\mathcal{F}_{\text{senc}}^{\text{unauth}}$, a symmetric encryption functionality with long-term keys $\mathcal{F}_{\text{ltsenc}}^{\text{unauth}}$ and a public key encryption functionality $\mathcal{F}_{\text{pke}}$.

As we do neither use the symmetric encryption functionality with long-term keys in our functionalities nor offer any way for the environment to access that functionality, we can implement the protocol without realizing this functionality, as it receives no messages at all.

4.5.4.1. Symmetric Encryption $\mathcal{F}_{\text{senc}}^{\text{unauth}}$

In [KT09b], Küsters and Tuengerthal prove that, in general, any IND-CCA2 secure symmetric encryption scheme can be used to implement the symmetric part of $\mathcal{F}_{\text{ENC}}$, but there are a few restrictions, which we explain first.

In general, $\mathcal{F}_{\text{senc}}^{\text{unauth}}$ cannot be implemented due to the *commitment problem* (for details, see, e. g., [BP04, CF01]): Consider some protocol that uses the ideal symmetric encryption functionality to encrypt some plaintext x under some freshly-generated and uncorrupted key k, resulting in some ciphertext y.

If the protocol not only sends the plaintext x and the ciphertext y to the adversary, but also reveals the key k to the adversary (e. g., by encrypting it under a corrupted key), the adversary may be able to decrypt y and, in the ideal world, detect that the decryption does not match x (but the leakage of x), while in the real world, y correctly decrypts into x. Thus, the adversary would able to distinguish both worlds.

In addition, key cycles (e. g., encrypting a key k_1 under a key k_2 and vice versa, see, e. g., [AR02]) also pose a problem as the usual security definitions are not able to cope with that kind of situations.

Therefore, in [KT09b], Küsters and Tuengerthal do only prove that the symmetric encryption functionality in $\mathcal{F}_{\text{ENC}}$ can be securely realized by any IND-CCA2 secure symmetric encryption scheme as long as $\mathcal{F}_{\text{ENC}}$ is accessed by a functionality that *does not cause the commitment problem* and that is *used order respecting*:

Roughly speaking, used order respecting protocols do not encrypt a key after it has been used in an encryption operation, which is a simple and natural restriction implying that no key cycles occur.

Our functionalities adhere to those restrictions, i. e., they do not encrypt a key after it has been used in an encryption operation and they never reveal keys that are not already known to the adversary:

The functionalities offer no interface for the environment or another functionality to directly use the encryption functionality or obtain keys, so we only have to argue that the operations performed by our functionalities themselves are used order respecting and do not cause the commitment problem.

First, our functionalities are used order respecting: The only key (of $\mathcal{F}_{\text{senc}}^{\text{unauth}}$) that is encrypted by another key is the symmetric key which is referred to by *ptr* in step (B.25); and *ptr* is encrypted only once, namely under pk^{ae}, before any plaintext is encrypted under *ptr*.

Then, our functionalities do not cause the commitment problem: There is no operation which uses a key for encryption that can *later* get known to the adversary. The adversary may corrupt both asymmetric as well as symmetric keys upon generation, and if an adversary corrupted a server's public key, each symmetric key that is encrypted with that corrupted key in step (B.25) is marked "known" (added to $\mathcal{K}_{\text{known}}$ by $\mathcal{F}_{\text{ENC}}$). But once any of these key is used for encryption, the "known" status is fixed for the rest of the execution.

Thus, using [KT09b, Theorem 10], we are able to substitute $\mathcal{F}_{\text{senc}}^{\text{unauth}}$ in $\mathcal{F}_{\text{ENC}}$ by a realization that uses any IND-CCA2 secure symmetric encryption scheme using a wrapper $\mathcal{P}_{\text{senc}}$ from using [KT09b].

4.5.4.2. Public Key Encryption $\mathcal{F}_{\text{pke}}$

The public key encryption functionality $\mathcal{F}_{\text{pke}}$ in [KT09b] is a slightly simplified version of the functionality $\mathcal{F}_{\text{pke}}$ defined in [KT08b]. As shown in [KT08b, Theorem 7] (and also stated in [KT09b, Theorem 7]), the functionality can be realized by a IND-CCA2 secure public key encryption scheme (again, without auxiliary input) using a straight-forward wrapper $\mathcal{P}_{\text{pke}}$.

4.5.5. The Key Store Functionalities $\mathcal{F}_{\text{KS}^{\text{sig}}}$ and $\mathcal{F}_{\text{KS}^{\text{ae}}}$

Our key store functionalities serve not only as a technical tool to simplify access to public keys (e. g., by initializing the key only once upon first access to a key), but also offer the clients and servers a method of fetching a party's public key from a *trusted* source:

The adversary can prevent the key store functionalities from delivering keys to other functionalities, but it cannot influence which key is sent (without corrupting the corresponding signature or encryption scheme).

Hence, any implementation would have to include some form of *trust model*. As discussed in Section 2.1.3.7, there are multiple methods for deciding if a key is trusted, but the methods usually involve some physical interaction to check the (physical) identity of a party that wants to establish trust in its key.

Therefore, we do not give a realization for the key store but argue that in real applications, one can implement both versions of $\mathcal{F}_{\mathrm{KS}}$ using standard techniques for building a public key infrastructure: In an implementation, the key store could be a local subroutine which,

1. locally stores and manages a single public/private key pair (which's public key has been published to a key server), and,

2. when requested to retrieve the public key of another party, fetches that key from a key server and locally checks its validity by using a trust model, e. g., a predefined set of certification authorities.

We note that one would assume that in such a scenario, the adversary could block access to a key server, which is modeled in our key store functionality in that the adversary may block any user from obtaining a key from the key store.

4.5.6. The Local Clock Functionality $\mathcal{F}_{\mathrm{LC}}$

On the one hand, the local clock functionality $\mathcal{F}_{\mathrm{LC}}$ over-approximates the powers of a realistic adversary: usually, one would not assume that the adversary can manipulate the local clock of a user before each access to that clock.

On the other hand, if the adversary is given access at all, our monotonicity restriction may be too strict.

Thus, there is no single natural implementation for this functionality; and as the IITM framework does abstract from "real time", there is no single "realistic" implementation inside the IITM framework. Therefore, we do not give a realization for $\mathcal{F}_{\mathrm{LC}}$, however, we discuss several aspects of possible implementations that are easy to prove secure.

First, note that it is possible to use the ideal functionality itself as an implementation as it features no unrealistic messages etc.

But it is also easy to replace the local clocks by one clock per party or even a synchronized global clock: Roughly speaking, a single instance of $\mathcal{F}_{\mathrm{LC}}$ could be accessed by all sessions of one party or even by all parties through a wrapper that synchronizes the clocks; and a simulator for proving that a party's clock implements $\underline{\mathcal{F}_{\mathrm{LC}}}$ or that a global clock implements $\underline{\underline{\mathcal{F}_{\mathrm{LC}}}}$ would simply maintain a single clock state and distribute it to the local clocks upon their requests.

It would also be possible to restrict the adversary's access (both for individual local clocks or synchronized clocks), e. g., to only set the initial value of the clock; the value could then advance each time that the functionality is activated, e. g., by a fixed or a randomly chosen value.

4.5.7. Implementing the Protocols

Theorem 4.1 now yields the following corollary:

Corollary 4.2. *Let p_{sig_1}, p_{sig_2}, p_{st}, p_{lt}, and p_{ae} be polynomials, let leak be a leakage algorithm that leaks exactly the length of a message, let Ω be an EUF-CMA secure signature scheme that is bounded by p_{sig_1}, let Σ_{se} be an IND-CCA2 secure symmetric encryption scheme that is bounded by p_{st}, let Σ_{ae} be an IND-CCA2 secure public key encryption scheme that is bounded by p_{ae}. Let $\mathcal{P}^{\mathrm{JS}'}_{\mathrm{SIG}}$, $\mathcal{P}'_{\mathrm{SIG}}$, $\mathcal{P}_{\mathrm{senc}}$, and $\mathcal{P}_{\mathrm{pke}}$ be the (systems of) IITM's referenced above. Define the following systems of IITM's:*

$$\widehat{\mathcal{P}}_{\mathrm{SIG}} = \mathcal{P}^{\mathrm{JS}'}_{\mathrm{SIG}}(\mathcal{T}_{\mathrm{sig}}, \mathcal{T}_{\mathrm{ver}}, p_{\mathrm{sig}_1}, p_{\mathrm{sig}_2}) \mid \underline{!\mathcal{P}'_{\mathrm{SIG}}(\Omega, \mathcal{T}_{\mathrm{sig}}, \mathcal{T}_{\mathrm{ver}})} \tag{4.16}$$

$$\widehat{\mathcal{P}}_{\mathrm{ENC}} = \mathcal{P}_{\mathrm{senc}}(\Sigma_{\mathrm{se}}) \mid \mathcal{F}_{\mathrm{ltsenc}}(p_{\mathrm{lt}}, \mathit{leak}, \hat{\mathcal{T}}^{\mathrm{lt}}) \mid \underline{!\mathcal{P}_{\mathrm{pke}}(\Sigma_{\mathrm{ae}}, \hat{\mathcal{T}}^{\mathrm{pke}})} \tag{4.17}$$

$$\widehat{\mathcal{P}}^{\mathrm{SA}}_{\mathrm{S2ME}} = !\mathcal{P}^{\mathrm{SA}}_{\mathrm{C}} \mid !\mathcal{P}^{\mathrm{SA}}_{\mathrm{S}} \mid \widehat{\mathcal{P}}_{\mathrm{SIG}} \mid \underline{\underline{!\mathcal{P}^{\mathrm{dummy}}_{\mathrm{SI}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{LC}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}}} \tag{4.18}$$

$$\widehat{\mathcal{P}}^{\mathrm{CSA}}_{\mathrm{S2ME}} = !\mathcal{P}^{\mathrm{CSA}}_{\mathrm{C}} \mid !\mathcal{P}^{\mathrm{CSA}}_{\mathrm{S}} \mid \widehat{\mathcal{P}}_{\mathrm{SIG}} \mid \widehat{\mathcal{P}}_{\mathrm{ENC}} \mid \underline{\underline{!\mathcal{P}^{\mathrm{dummy}}_{\mathrm{SI}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{LC}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}^{\mathrm{ae}}}}} \tag{4.19}$$

$$\widehat{\mathcal{P}}^{\mathrm{PA}}_{\mathrm{S2ME}} = !\mathcal{P}^{\mathrm{PA}}_{\mathrm{C}} \mid !\mathcal{P}^{\mathrm{CSA}}_{\mathrm{S}} \mid \widehat{\mathcal{P}}_{\mathrm{SIG}} \mid \widehat{\mathcal{P}}_{\mathrm{ENC}} \mid \underline{\underline{!\mathcal{P}^{\mathrm{dummy}}_{\mathrm{SI}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{LC}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}}} \mid \underline{\underline{!\mathcal{F}_{\mathrm{KS}^{\mathrm{ae}}}}} \mid \mathcal{F}_{\mathrm{RO}} \tag{4.20}$$

Then, we have:

$$\widehat{\mathcal{P}}^{\mathrm{SA}}_{\mathrm{S2ME}} \leq^{\text{BB-noaux}} \mathcal{F}_{\mathrm{S2ME}}(\mathit{leak}_{\mathrm{full}}, \mathtt{false}) \ , \tag{4.21}$$

$$\widehat{\mathcal{P}}^{\mathrm{CSA}}_{\mathrm{S2ME}} \leq^{\text{BB-noaux}} \mathcal{F}_{\mathrm{S2ME}}(\mathit{leak}_{\mathrm{length}}, \mathtt{false}) \ , \tag{4.22}$$

$$\widehat{\mathcal{P}}^{\mathrm{PA}}_{\mathrm{S2ME}} \leq^{\text{BB-noaux}} \mathcal{F}_{\mathrm{S2ME}}(\mathit{leak}_{\mathrm{full}}, \mathtt{true}) \ . \tag{4.23}$$

Proof. The corollary follows from Theorem 4.1 by 1. Theorems 5, 6 and 7 from [KT08b], 2. Theorem 10 and Corollary 3 from [KT09b], 3. the above reasoning that our functionalities do not cause the commitment problem and are used order respecting, and 4. a trivial simulator for $\mathcal{P}^{\mathrm{dummy}}_{\mathrm{SI}}$. □

4.6. Comments and Caveats

In this section, we comment on some aspects of the IITM framework or simulation-based security paradigms.

4.6.1. Roles of the Environment and the Adversary

When designing functionalities, one gains insights in the multiple roles that the environment and the adversary play.

The main role that the environment is supposed to model is the role of the layer *above* the functionalities under analysis; for example, this may be a set of programs or another protocol layer. But the environment also has to provide resources to the functionalities on enriching tapes, and it has to check the corruption status of the functionalities (see below on notes why this is necessary).

The adversary also plays at least three roles: 1. Routing (and possibly manipulating) the network traffic, 2. corrupting functionalities and taking over their operation, and

3. serving as an universal quantifier for parameters (see, e. g., step (B.33), or the signature functionality $\mathcal{F}_{\mathrm{SIG}}$, where the adversary may determine the signature scheme).

Especially because of the resources, the modeling of functionalities is, in part, unnatural.

For example, we included the signature interface $\mathcal{P}_{\mathrm{SI}}$ in $\mathcal{P}_{\mathrm{S2ME}}$ to give the adversary partial access to the signature functionality in the implementation, but naturally, we wanted to leave that out of the ideal functionality (which may be realized by a protocol that does not use digital signatures at all). But the signature functionality needs resources from the environment, so $\mathcal{P}_{\mathrm{SI}}$ has an enriching input tape from the environment.

Thus, we have to equip $\mathcal{F}_{\mathrm{S2ME}}$ with a corresponding tape, which is realized in the enriching input functionality $\mathcal{F}_{\mathrm{EI}}$. But $\mathcal{F}_{\mathrm{EI}}$ is not only an artificial addition to the ideal functionality, it is also too generic to easily evaluate its consequences—this is undesirable for an ideal functionality, which is the equivalence of a "security definition" in more specific models (like the Bellare–Rogaway framework in Section 3).

Similarly, if one wants to really implement a realization given in this thesis, one has to closely distinguish which messages sent to the adversary and expected from the adversary only concern corruption and universal quantification (see above for examples). For corruption messages is seems obvious what has to be dropped; but, e. g., one also has to drop questioning the adversary for parameters in step (B.33), and dropping or changing messages may always have non-trivial security implications.

The multiple roles of environment and the adversary are also referenced when talking about *correctness* of protocols in simulation-based frameworks, see next section.

4.6.2. Correctness Definition

In [BR93a], Bellare and Rogaway did not only define security notions for protocols, but also correctness notions (which we added to our setting in Section 3.4.1); in contrast, simulation-based frameworks lack a strict definition for correctness of protocols.

This is especially interesting because the security definition allows any functionality $\mathcal{F}$ to be securely realized by a modification $\mathcal{F}^{'}$ of that $\mathcal{F}$ that does not send out any messages to the adversary (the simulator that proves that $\mathcal{F}^{'}$ realizes $\mathcal{F}$ simply blocks all communication between the $\mathcal{F}$ and the adversary), cf. the notion of a *non-trivial protocol* in [CLOS02]; this is similar to other security definitions where any protocol that does not send any message at all is secure in the sense that it leaks no information at all.

But it is also possible to define realizations (and prove them secure) that only partially implement a given ideal functionality. For example, our functionalities $\mathcal{F}_{\mathrm{S2ME}}$ could be securely realized by implementations that only deliver requests, but never deliver responses; a simulator would then simply block all communication between the adversary and the ideal functionality related to response messages.

These are examples where it is obvious from the realization that something is "missing", but there may be more subtle examples.

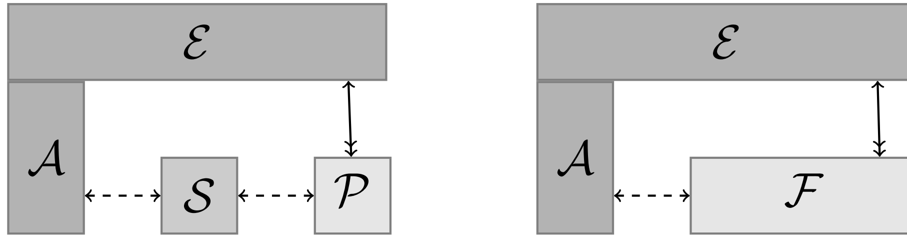

Figure 4.5.: A hypothetical approach for a correctness definition in the IITM framework

So if one wants to define correctness in the IITM framework, an approach that comes to mind is to swap the roles of ideal and real functionalities: As illustrated in Figure 4.5, the adversary would have direct access to the ideal functionality $\mathcal{F}$ and may start any number of instances of the ideal functionality, while the simulator has to use the real system $\mathcal{P}$ to produce results that are indistinguishable from the ideal world. Thus, we would be able to guarantee that the real protocol works as the ideal functionality in all but a negligible number of cases.

In mathematical terms, we would simply swap the roles in the definition of security and say that a real system $\mathcal{P}$ *correctly implements* an ideal functionality $\mathcal{F}$ if there is a simulator $\mathcal{S}$ such that for all adversaries $\mathcal{A}$ and environments $\mathcal{E}$ for $\mathcal{S} \mid \mathcal{P}$ or $\mathcal{F}$, the systems $\mathcal{E} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{P}$ and $\mathcal{E} \mid \mathcal{A} \mid \mathcal{F}$ are computationally indistinguishable.

If we then prove that a realization correctly implements an ideal functionality, we know that it is not a partial implementation (nor an implementation that does not send any messages at all, see above).

Note that, naturally, the composition results would transfer to the case of correctness, i. e., if a protocol uses several ideal functionalities, we can use a modular proof to show that the protocol is a correct implementation.

This definition would also force us to model ideal functionalities closer to what is realizable, so we would over-approximate less: Consider a functionality like $\mathcal{F}_{\text{SIG}}$ that precisely models the abilities of an adversary to corrupt that functionality. Now if we define an ideal functionality that is implemented by some realization that uses $\mathcal{F}_{\text{SIG}}$ (e. g., when we define an ideal functionality like $\mathcal{F}_{\text{S2ME}}$), it is easy to model corruption in $\mathcal{F}_{\text{S2ME}}$ by the corruption macro introduced in [KT08b] (as done in [KSW09b, KSW09c]). But this over-approximates the abilities of the adversary upon corruption, as it allows it to send arbitrary messages to the environment. In contrast, in this thesis, the modeling of corruption in $\mathcal{F}_{\text{S2ME}}$ is more complex, but also more precise, and thus, we would be closer to developing a provable correct implementation of the ideal functionality.

But this definition does not yield a natural definition for "correctness" since it is easy to define protocols which are "correct" in this sense, but not in an intuitive sense:

Take any protocol $\mathcal{P}$ that securely and correctly implements an ideal functionality $\mathcal{F}$. We then define a simple variation $\mathcal{P}^*$ that appends the bit 0 to each outgoing message, but expects that each incoming message ends with 1 and strips this last bit before pro-

cessing the message like $\mathcal{P}$ does. Now $\mathcal{P}^*$ also securely *and* correctly implements $\mathcal{F}$, as both simulators may append and strip these bits as necessary. But calling $\mathcal{P}^*$ a correct protocol is unnatural, as it relies on the network to change messages before delivery. So, the definition only guarantees that a realization is *as powerful* as an ideal functionality in some sense.

In addition, for functionalities that expect parameters from the adversary (like $\mathcal{F}_{\text{SIG}}$, which expects to receive the algorithms for the signature scheme from the adversary), our potential correctness definition leaves it to the simulator to decide which parameters to chose. Thus, parameters that are universally quantified in the security definition become existentially quantified in the correctness definition.

When analyzing the situation more closely, we observe that by distinguishing between the roles of the adversary (network, corruption, universal quantification), we see that for correctness, we would usually want to treat these three roles of the adversary (which are then taken on by the simulator) differently:

- We would want to limit the network part to deliver messages to the correct recipient, but possibly quantify over all interleavings of messages of different sessions etc. This is what is done with the benign adversary in Section 3.
- The simulator should be free to simulate any corruption done in the ideal world by the adversary, so we would not want to restrict the corruption part.
- For parameters, we usually would not want existential quantification (which would only prove that the protocol works correctly for a fixed set of parameters), and perhaps nor universal quantification (e. g., in the case where the parameter is a signature scheme as in $\mathcal{F}_{\text{SIG}}$) either, but instead, e. g., quantification over a fixed set.[15]

We remark that we believe that the results that could be obtained from this (a more precise definition of correctness) do not justify the complexity added by explicitly distinguishing these types of communication with the adversary (or the simulator in case of correctness).

4.6.3. Technicalities

When modeling (systems of) IITM's, one has to take care of a few technical, but non-trivial tasks.

4.6.3.1. Resources

When modeling systems of IITM's, one has to conform to resource restrictions, i. e., the systems have to be well-formed and the length of output messages is restricted by the

[15] This could be achieved by parameterizing the functionalities and then stating that for each choice of parameters fulfilling some restriction, the protocol could be proven correct; however, in some situations (like the choice of protocol parameters in step (B.33)), one wants to prove the protocol correct even if the parameters are chosen *per identity*.

length of input received on enriching tapes.

Our functionality $\mathcal{F}_{\text{MX}}$ receives its request payload on an enriching tape on the client side, so it could deliver its payload to the environment on the server side without breaking resource restrictions. However, when realizing the functionality, the server receives the payload on a network tape, i. e., a consuming tape. Thus, it cannot deliver payloads of arbitrary length to the environment, but has to use a buffer size (see variable n in the versions of $\mathcal{P}_{\text{S}}$). To make both worlds indistinguishable, we have to also introduce this concept to the ideal world. But as the instances of $\mathcal{F}_{\text{MX}}$ do not connect to the environment on the server side before the payload is delivered, we need another mechanism to receive resources. Thus, we introduced $\mathcal{F}_{\text{SM}}$, which manages the resources per identity and allows instances of $\mathcal{F}_{\text{MX}}$ to "fetch" resources when necessary.

Now, at first, one would like to fetch only so much resources as necessary. But $\mathcal{F}_{\text{MX}}$ has no way of signaling $\mathcal{F}_{\text{SM}}$ how much resources are necessary: Even a message that only contains the length of the payload has a length that depends on the payload itself. So, $\mathcal{F}_{\text{SM}}$ always sends *all* available resources to $\mathcal{F}_{\text{MX}}$.

This could be avoided by integrating $!\underline{\mathcal{F}_{\text{MX}}} \mid \mathcal{F}_{\text{SM}}$ into a single IITM or, as in [KSW09b, KSW09c], by splitting client and server in the ideal world, but the first approach reduces the intended modularity of the functionality and the second approach adds complexity to the message transfer itself.

4.6.3.2. Addressing

Addressing (instances of) IITM's is not a trivial task either: Multiple instances of a single IITM functionality $\mathcal{M}_0$ may run in parallel (using the bang operator), and while the CheckAddress mode of the IITM framework mostly allows to clearly define which instance accepts which message, a general problem in simulation-based frameworks still occurs:

Suppose that multiple (instances of) IITM's $\mathcal{M}_1, \mathcal{M}_2, \ldots$ want to access one of multiple instances of $\mathcal{M}_0$ that run in parallel. Then they have to use some identifying string like a session id to address a single instance, for example, $\mathcal{P}_{\text{C}}$ and $\mathcal{P}_{\text{S}}$ both access a single instance of $\underline{\underline{\mathcal{F}_{\text{SIG}}}}$ using identifiers like $(s, (\mathsf{S}, c, r))$. For the realizations in this thesis, agreeing on a globally unique session identifier is easy to realize, as the client can simply chose the nonce r at random and include it in the message to the server.

For a message exchange functionality like $\mathcal{F}_{\text{MX}}$, the situation is different: As shown above, $\mathcal{F}_{\text{MX}}$ has to obtain resources from the environment on the server side before delivering a request message to the environment. Now if we assume that we leave out $\mathcal{F}_{\text{SM}}$ and if the client would like to send a message and the server would like to open a "buffer" for exactly that incoming message, both parties would have to agree on a session id *before* an instance of $\mathcal{F}_{\text{MX}}$ could transfer messages between both parties.

But how can one agree on a session id before the protocol is run? In [Can00, Section 3.4.2], Canetti states in a paragraph "on determining the session identifiers":

> There are multiple ways for such agreement to take place. A first method [...] is to determine the SID of the instance in advance [...] A second alter-

> native is to design the protocol in such a way that the agreement on the SID is done by the protocol instance itself. [...] A third alternative [...] is to run some simple agreement protocol among the parties [...]

The first and the third approach would require communication before starting the protocol, while the second approach is roughly what we use in our functionalities[16], but this has the drawback that, as discussed in Section 4.6.3.1, it only works with a long-lived functionality like $\mathcal{F}_{\mathrm{SM}}$ or $\mathcal{P}_{\mathrm{S}}$ which spans multiple sessions.

4.6.3.3. Corruption

Corruption refers to the adversary's ability to selectively take over (parts of) functionalities, which intends to model situations where, e. g., parties of a protocol (partially) cooperate with the adversary or where a party mistakenly trusts the adversary.

Usually, if an instance of a functionality is corrupted, nothing is guaranteed for the security of (a part of) that instance; but the security guarantees for other instances may still hold (precisely defining this is a part of the ideal functionality).

We stress that it is important to enable the environment to always check if a functionality is corrupted or not (as mentioned in [KT08a]), otherwise writing a simulator and thus proving a realization secure is a trivial task: If we would have left out step (4.6.5) and similar steps, a simulator could corrupt each instance of $\mathcal{F}_{\mathrm{MX}}$ and deliver arbitrary payloads; thus, an insecure protocol could be proven secure.

We also stress that, when using ideal functionalities (like we used $\mathcal{F}_{\mathrm{SIG}}$ etc.), one has to closely examine the corruption functionalities of those ideal functionalities: As the abilities of an adversary to corrupt such a functionality are modeled after realistic assumptions (like the adversary obtaining the private key of a party), one has to accept that such a functionality may get corrupted—if, in contrast, one stops cooperating with such a functionality (like we did in [KSW09b, KSW09c], see Section 4.6.5), one only proves the protocol secure for the case that no corruption occurs; thus, one has no guarantees for the case that, e. g., a single key is corrupted.

Some functionalities need additional resources when corrupted, this is reflected by the corruption macro introduced in [KT08b] (also see Section 5.1.1, where we use that macro in an ideal functionality): Here, a corrupted functionality needs to be provided with resources to allow the adversary to send messages through that functionality.

But this simple mechanism for distributing resources poses a problem if multiple corrupted functionalities are involved: Assume for example that a functionality $\mathcal{P}$ uses multiple instances of the signature functionality $\mathcal{F}_{\mathrm{SIG}}$. Now if the environment sends resources for corruption to $\mathcal{P}$, these resources have to be distributed to all the corrupted ideal functionalities, but in each case, the adversary would have to be notified of the resources that have been sent. But the adversary cannot be forced to give back control

[16] In $\mathcal{F}_{\mathrm{S2ME}}$, the functionality $\mathcal{F}_{\mathrm{MX}}$ chooses SID's for the server and the adversary; in $\mathcal{P}_{\mathrm{S2ME}}$, the client chooses a nonce which then identifies the session.

after it has been notified that the first of the corrupted functionalities received its resources, so we may have to, e. g., force it to give back control or reduce the number of notifications (which may result in decreased modularity).

4.6.4. Joint State Realizations

The IITM framework features a simple and natural mechanism for IITM's with joint state, see [KT08a].

But the joint state realization from [KT08a] that we used in Section 4.5 to implement $\mathcal{F}_{\mathrm{SIG}}$ has the drawback that it adds unnatural prefixes to messages: In a joint state implementation, a single instance of a real signature scheme is used to realize a multi-session version of $\mathcal{F}_{\mathrm{SIG}}$ by adding the session id to the message before signing, i. e., if a message m is signed in a session identified by sid, the term (sid, m) is passed to the signature algorithm; this is usually unrealistic.

Note that in [GMP+08], where the security protocol TLS [DA06] is analyzed, the joint-state theorem for the Universal Composition framework is used without taking into account that the resulting implementation would prefix messages before encrypting them; thus the authors proved secure a variant of TLS that is not used in applications.

In our case, this modeling even reduces the abilities of the adversary in comparison with Chapter 3: The adversary does not have the ability to pass arbitrary bit strings (that do not look like messages) to the signature scheme, as the signature scheme only receives bit strings that have the structure (sid, m) with $sid, m \in \{0,1\}^*$.

For our protocols, these prefixes are also unnecessary as they only contain data that is already extractable from our messages (sender, receiver and message id, which are included in the header of our messages). Thus, we could have used just one instance of the signature scheme per identity (i. e., $!\underline{\mathcal{F}_{\mathrm{SIG}}}$ instead of $!\underline{\underline{\mathcal{F}_{\mathrm{SIG}}}}$). We sketch the modifications to our functionalities and the proof.

The functionalities $\mathcal{P}_{\mathrm{C}}$ and $\mathcal{P}_{\mathrm{S}}$ (as well as $\mathcal{P}_{\mathrm{SI}}$ and $\mathcal{F}_{\mathrm{KS}^{\mathrm{sig}}}$) would have to drop the second prefix, e. g., they would communicate with the signature functionality by using prefixes $(c, \ldots$ and $(s, \ldots$ instead of $(c, (\mathrm{C}, s, r), \ldots$ etc.

The simulators would have to be modified accordingly, but one would also have to change the handling of corruption: If a signature functionality is corrupted, *all* sessions that use the corrupted signature functionality would have to be corrupted, and as soon as a new session is started, one would have to check if one has to corrupt that session.

In our proof, for step (B.22) we used the fact that the signature functionality signed only one request message; this could simply be modified to state that the signature functionality signed only one request message containing these values for sender, receiver and message id in its header (with overwhelming probability). Analogous modifications would have to be made to the proof for step (B.38) and the corresponding steps for CSA and PA.

While these modifications would be possible, it seems more natural from a modeling perspective to use different instances of the signature functionalities for different ses-

sions. To preserve this modeling, one could use an alternative approach: As the session id can be computed from a message in our work, we can partition the message space.

We sketch a joint state realization $\mathcal{P}^{\text{JS}*}_{\text{SIG}}$ such that, roughly, $\mathcal{P}^{\text{JS}*}_{\text{SIG}} \mid \mathcal{F}_{\text{SIG}}$ realizes $!\underline{\mathcal{F}_{\text{SIG}}}$. Our realization $\mathcal{P}^{\text{JS}*}_{\text{SIG}}$ would be parameterized by an efficiently computable function $f\colon \{0,1\}^* \to \{0,1\}^*$ that partitions the message space, i. e., on input of a message, f returns the session id of a session to which that message seems to belong, or ε if no session could be identified. Now whenever $\mathcal{P}^{\text{JS}*}_{\text{SIG}}$ receives a message containing a session id *sid* and the request to sign or verify a message m, the functionality checks whether $f(m)$ equals *sid*. If both are equal, the request is forwarded to the one instance of $\mathcal{F}_{\text{SIG}}$ used by $\mathcal{P}^{\text{JS}*}_{\text{SIG}}$; otherwise, the request is simply answered by returning $\perp$ as the signature or the verification bit.

To show that $\mathcal{P}^{\text{JS}*}_{\text{SIG}} \mid \mathcal{F}_{\text{SIG}}$ securely realizes $!\underline{\mathcal{F}_{\text{SIG}}}$, we use a simulator that is similar to $\mathcal{S}^{\text{JS}}_{\text{SIG}}$ in [KT08b], which asks the adversary once to provide algorithms s and v and then uses variants s_{sid}, v_{sid} each time an instance of $\underline{\mathcal{F}_{\text{SIG}}}$ with session id *sid* asks for it. The main modification here is that s_{sid} and v_{sid} would be defined to first run f on the message and then return $\perp$ if $f(m) \neq sid$, and otherwise execute s or v.

Using this realization, we could realize $!\underline{\underline{\mathcal{F}_{\text{SIG}}}}$ in such a way that only one instance of $\mathcal{F}_{\text{SIG}}$ is used per party, but our ideal modeling could be preserved. Our partitioning function f would simply check if the message has the structure of a request or response message, and return ε if not. Otherwise, f extracts and returns the appropriate triple, e. g., (C, s, r).

Note that while our functionalities $\mathcal{P}_{\text{C}}$ and $\mathcal{P}_{\text{S}}$ would respect our partitioning function f when signing, i. e., direct requests for signing to the "correct" instance of the signature functionality, the signature interface $\mathcal{P}_{\text{SI}}$ only allows the adversary to sign any message that has *not* the structure of a request or response. Thus, the adversary simply has to use the instance of $\mathcal{P}_{\text{SI}}$ identified by session id ε for access to the signature scheme, but (contrary to the current joint-state realization, see above) this allows the adversary to pass arbitrary bit strings (that do not look like messages) to the signature scheme.

4.6.5. Comparison with Pre-Published Results

Parts of the results in this chapter were previously published in [KSW09b, KSW09c]. But even for these parts, there were some differences, which we point out below.

In [KSW09b, KSW09c], we only analyzed a different modeling of SA2ME-1 (called 2AMEX-1 as noted in Section 2.5.1), but neither CSA2ME-1 nor PA2ME-1. The latter protocols have neither been formally defined nor analyzed before in published work. Hence, the ideal functionality in [KSW09b, KSW09c] differs in that it is rewritten in this work to be compatible not only with SA2ME-1, but also with CSA2ME-1 and PA2ME-1.

The functionalities for SA2ME-1 in this thesis differ from those in [KSW09b, KSW09c] for 2AMEX-1 in other aspects:

In this thesis, the modeling of corruption is far more detailed:

In [KSW09b, KSW09c], we only used a pre-defined corruption macro from [KT08a, KT08b], but we did not model that the environment could pass resources to corrupted signature schemes, and once a signature scheme was corrupted, we essentially stopped working with that signature scheme. This essentially disables the adversaries ability to corrupt the signature schemes.

In this thesis, we correct that flaw by allowing the adversary to corrupt signature and verification functionalities (as well as encryption functionalities) and also allowing the environment to pass resources to the signature functionalities, which the adversary needs to properly use the corrupted schemes. In addition, we cooperate with signature or encryption schemes even if they are corrupted, but include that information in the answer to the environment's question for a functionality's corruption status.

In [KSW09b, KSW09c], we modeled the ideal functionality in such a way that client and sever sides were separated. While this would also be possible in our current modeling, we feel that the modeling in this thesis is more natural, as the message transfer is handled by a single instance of $\mathcal{F}_{\text{MX}}$, holding all values and state information of one run of the protocol.

This also allows for a more natural modeling of the expiration of sessions on the server side: While in [KSW09b, KSW09c] the adversary was able to provoke error messages at any time (even for non-expired sessions), it now only has the ability to irrevocably expire a session, and $\mathcal{F}_{\text{MX}}$ then consistently handles the error messages.

On the server side, we now use a session id sid_s that is chosen by the server independent of the nonce r used in the protocol messages. In contrast, the modeling in [KSW09b,KSW09c] has the following disadvantage: Suppose a message m contained some message id r, and suppose that m was sent by the client, but not yet received by the server. Now the adversary could read r and pass it on to the environment, which could then try to respond to a message before the server even received that message.

A similar modification that simplifies the modeling is that on the client side, the environment now chooses its own session id (sid_c), whereas in [KSW09b, KSW09c], the generated nonce r was used by the environment to distinguish sessions on the client side, what made additional protocol steps necessary.

In summary, while the general result for SA2ME-1 is the same in both in [KSW09b, KSW09c] and in this thesis, the modeling of the ideal functionality is more realistic, precise, and flexible in this thesis.

5. Relation between the Two Frameworks

In this chapter we study the relation between the Bellare–Rogaway framework and the IITM framework.

First, in Section 5.1, we show that for a simpler case of protocols (mutual authentication protocols), there is a strong connection between both frameworks: We define a lightweight wrapper that turns any mutual authentication protocol that is secure as defined in [BR93a] (and has only polynomially many rounds) into a system of IITM's that implements a standard ideal functionality for authentication in simulation-based frameworks.

For the case of secure two-round message exchange, we then make some remarks about the relation between both models in Section 5.2.

5.1. Mutual Authentication

In this section we show that a mutual authentication protocol (with polynomially many rounds) secure in the Bellare–Rogaway framework implements ideal mutual authentication. To this end we describe two variants of an ideal functionality for mutual authentication in the IITM framework, a single-session and a multi-session version, and give realizations of both ideal functionalities by using any protocol secure in the Bellare–Rogaway framework.

The single-session variant of the ideal functionality is the direct adaptation of the functionality defined in [CH06] for mutual authentication, which is intended to model the simplest possible case of authentication. The multi-session variant is a more practical modeling, see our remarks below. Since both variants are similar, we concentrate on the more complex multi-session variant later on.

In Section 5.1.4 we later show that the opposite direction does not hold, i. e., a protocol secure in the IITM framework does not naturally yield a protocol secure in the Bellare–Rogaway framework. In addition, as explained in Section 4.6.2, it is hard to capture the correctness definition of the Bellare–Rogaway framework in the IITM framework.

5.1.1. Ideal Functionalities

The first variant of the ideal functionality of mutual authentication, $\mathcal{F}_{\mathrm{MA}}^{\mathrm{SS}}$, defined in Appendix C.1.1, is a direct adaption of the authentication functionality $\mathsf{F}_{2\mathrm{MA}}$ from [CH06] to the IITM framework: To initiate authentication, a party sends its own identity as well as the identity of the intended communication partner to the ideal functionality. If

the ideal functionality receives two matching tuples (i, j) and (j, i) and as soon as the adversary allows it, the functionality informs the two parties of the successful authentication.

We also want to cover the multi-session case, where the same pair of partners can perform multiple runs of the protocol in parallel. The single-session variant $\mathcal{F}_{\mathrm{MA}}^{\mathrm{SS}}$ could be extended to a multi-session version $\underline{\mathcal{F}_{\mathrm{MA}}^{\mathrm{SS}}}$ using the standard mechanism for multi-session extensions in the IITM framework (denoted by underlining the functionality, see Section 4.1.2). However, both the multi-session version $\underline{\mathcal{F}_{\mathrm{MA}}^{\mathrm{SS}}}$ and the functionality $\mathsf{F}_{2\mathrm{MA}}$ from [CH06] have the drawback that both communication partners have to agree on a session id *before* using the functionalities, which seems unnatural, see Section 4.6.3.2.

Therefore, we define a second variant of the ideal functionality, $\mathcal{F}_{\mathrm{MA}}^{\mathrm{MS}}$, see Figure C.1.2, which directly allows the users to initiate multiple sessions between the same communication partners: Each user employs local session id's to distinguish different sessions between the same communication partners, the session id's of both partners do not have to match. This is realistic and more closely represents the situation in the Bellare–Rogaway framework: The question which sessions exchange messages depends on the delivery of these messages by the network. To model this, in our ideal functionality $\mathcal{F}_{\mathrm{MA}}^{\mathrm{MS}}$, the adversary decides which session id's are partnered up, e. g., the adversary may connect a session of party i that wants to connect to party j and has a local session id s to a session running for party j that wants to connect to party i and has local session id $t \neq s$.

Note that while variant SS is a direct adaption of [CH06], the latter variant MS corresponds to the definition of protocols in [BR93a], where protocols are required to be secure even if the protocol itself does neither know if there are parallel sessions of the same protocol, nor is it provided with a (local) session id. We show that *both* variants are implemented by protocols that are secure in the Bellare–Rogaway framework.

Our functionalities use the parameterized corruption macro `Corr` (see Appendix C.4) that adds a couple of steps which take precedence over the steps we defined above. The corruption macro is a simple adaption of the one defined in [KT08b], we added parameters for working with message prefixes.

5.1.2. Implementing Mutual Authentication

In the following, we fix a secure mutual authentication protocol Π and a long-lived key generator $\mathcal{G}$, both as defined in Section 3.1.

For this work, we restrict ourselves to protocols with *polynomially many rounds*, i. e., there is a polynomial p such that when Π is run with security parameter η, then each session $\Pi_{i,j}^{s}$ sends at most $p(\eta)$ outgoing messages. Further, without loss of generality, because both Π and $\mathcal{G}$ have polynomial running time in η, we can assume that there is a polynomial q such that both algorithms only use the first $q(\eta)$ bits of their random bit string.

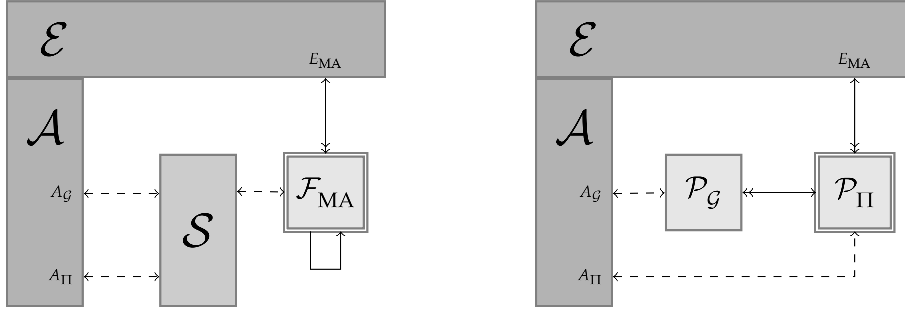

Figure 5.1.: Ideal world (left) and realization (right) for mutual authentication

To adapt the protocol to the IITM framework, for any $v \in \{\text{SS}, \text{MS}\}$, we define a system of IITM's $\mathcal{P}^v_{\text{MA}} = !\mathcal{P}^v_{\Pi} \mid \mathcal{P}^v_{\mathcal{G}}$ which securely implements $!\mathcal{F}^v_{\text{MA}}$, as illustrated in Figure 5.1.

The IITM's $\mathcal{P}^{\text{MS}}_{\Pi}$ and $\mathcal{P}^{\text{MS}}_{\mathcal{G}}$ are defined in Appendices C.2.2 and C.2.4, respectively; the functionalities for the simpler case of variant SS can be found in Appendices C.2.1 and C.2.3. In the following, we only describe variant MS.

The IITM $\mathcal{P}^{\text{MS}}_{\mathcal{G}}$ is a simple wrapper around $\mathcal{G}$. During initialization, it chooses a random bit string r from $\{0,1\}^{q(\eta)}$, where—as explained above—we assume that $q(\eta)$ is a polynomial that, for any security parameter η, gives the maximal number of bits that $\mathcal{G}$ or Π use of their random bit string. When the IITM is called with an identity a, it responds with the output of $\mathcal{G}(1^\eta, a, r_{\mathcal{G}})$. It also allows the adversary to retrieve the value of $\mathcal{G}(1^\eta, \mathcal{A}, r_{\mathcal{G}})$ (see Section 3.1).

When the machine $\mathcal{P}^{\text{MS}}_{\Pi}$ is first called with identities i and j and a session id s, it randomly chooses r from $\{0,1\}^{q(\eta)}$ and retrieves the private information of i from $\mathcal{P}^{\text{MS}}_{\mathcal{G}}$. Then, for every incoming message, $\mathcal{P}^{\text{MS}}_{\Pi}$ directly calls the algorithm Π with the incoming message m_{in} and additional information (security parameter 1^η, identities i and j, private information a, message trace κ, and random bit string r) and relays the response m_{out} to the network together with the decision δ. If the algorithm Π accepts (i. e., $\delta = \mathsf{A}$), the IITM allows the adversary to initiate notification of the environment about successful authentication.

As explained in Section 4.1.1, the accumulated length of messages that machines may print out during the entire protocol run is restricted polynomially in the security parameter and input received on *enriching* tapes. The machines in our protocol satisfy this requirement: For the key generator this is obvious as each of its outputs is triggered by an input on an enriching tape. The protocol machines $\mathcal{P}^v_{\Pi}$ also satisfy this condition: The entire output of $\mathcal{P}^v_{\Pi}$ is polynomial since 1. each protocol has only a polynomial number of rounds, 2. the messages produced by Π are bounded because the runtime of Π is polynomial in η, 3. the output message to the environment is only sent once, and 4. only one message is sent to $\mathcal{P}^{\text{MS}}_{\mathcal{G}}$.

5.1.3. Bellare–Rogaway Security Implies Secure Realization

Theorem 5.1. *Let* Π *be a secure and correct mutual authentication protocol with polynomially many rounds, and let* $\mathcal{G}$ *be a long-lived key generator, both as defined in [BR93a]. Then for both* $v = \mathrm{SS}$ *and* $v = \mathrm{MS}$ *we have*

$$!\mathcal{P}^{v}_{\Pi} \mid \mathcal{P}^{v}_{\mathcal{G}} \leq^{\mathrm{BB}} !\mathcal{F}^{v}_{\mathrm{MA}} \ . \tag{5.1}$$

Proof of Theorem 5.1. We only prove the more involved case $v = \mathrm{MS}$, the simpler case $v = \mathrm{SS}$ follows easily with a simplified proof.

We show that when we use the simulator $\mathcal{S}^{\mathrm{MS}}_{\mathrm{MA}}$ presented in Appendix C.3, it follows that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, with overwhelming probability the systems

$$\mathcal{M}_{\mathcal{F}} = \mathcal{E} \mid \mathcal{A} \mid !\mathcal{F}^{\mathrm{MS}}_{\mathrm{MA}} \mid \mathcal{S}^{\mathrm{MS}}_{\mathrm{MA}} \quad \text{and} \quad \mathcal{M}_{\mathcal{P}} = \mathcal{E} \mid \mathcal{A} \mid !\mathcal{P}^{\mathrm{MS}}_{\Pi} \mid \mathcal{P}^{\mathrm{MS}}_{\mathcal{G}} \tag{5.2}$$

result in the same output. More precisely, we show that for every possible sequence of random bits used by 1. the adversary, 2. the environment, 3. the (simulated or real) LL-key generator, and 4. the (simulated or real) protocol algorithms, the parts of the real and ideal systems that are visible to $\mathcal{E}$ or $\mathcal{A}$ behave identically with overwhelming probability. This implies that with overwhelming probability, the environment and the adversary produce the same output when interacting with the real and the ideal system, as required.

Hence let $\mathcal{E}$ and $\mathcal{A}$ be an arbitrary polynomial-time environment and adversary for our functionalities, respectively. In order to prove the result, we establish a relation R between states of the two systems, and show that if both systems use the same randomness as explained above, then the following holds with overwhelming probability: If q_1 is a state of $\mathcal{M}_{\mathcal{F}}$, and q_2 is a state of $\mathcal{M}_{\mathcal{P}}$ such that $(q_1, q_2) \in R$, then

1. the messages sent and received by $\mathcal{E}$ and $\mathcal{A}$ are identical in q_1 and q_2,
2. for every message m that $\mathcal{E}$ or $\mathcal{A}$ can send to the system, if q'_1 is the follow-up state of $\mathcal{M}_{\mathcal{F}}$ and q'_2 is the follow-up state of $\mathcal{M}_{\mathcal{P}}$, then $(q'_1, q'_2) \in R$.

Hence inductively for every sequence of actions that the coalition of $\mathcal{E}$ and $\mathcal{A}$ performs, the reactions they observe from the systems are identical as claimed above. The relation R essentially establishes a bisimulation (see [Par81]) between the two systems. We now make this more precise.

Definition of the relation R. In the following, with a *state* of either $\mathcal{M}_{\mathcal{P}}$ or $\mathcal{M}_{\mathcal{F}}$ we mean the configurations of all involved machines at either (a) the beginning of the protocol run, or (b) a time when $\mathcal{E}$ or $\mathcal{A}$ is activated, where a machine which is neither $\mathcal{E}$ or $\mathcal{A}$ was active before.

For states q_1 of $\mathcal{M}_{\mathcal{F}}$ and q_2 of $\mathcal{M}_{\mathcal{P}}$ as above, let $(q_1, q_2) \in R$ if and only if

- the communication history between $\mathcal{E}$ and $\mathcal{A}$ and the remaining machines is identical in q_1 and q_2, and

- the same machine is active in q_1 and q_2 (this must be either $\mathcal{E}$ or $\mathcal{A}$, note that $\mathcal{E}$ is active when a protocol run is started).

Proof of required properties. We show that R has the properties as mentioned above. The first property directly follows from the definition of R. It remains to show the second property, i. e., if $(q_1, q_2) \in R$, and the same message is sent to the system by $\mathcal{E}$ or $\mathcal{A}$, then the resulting states are again R-related. First observe that from the definition of R, the following properties hold:

- A machine $\mathcal{P}_{\Pi}^{\text{MS}}$ for the session (i, j, s) is running in the state q_2 of $\mathcal{M}_{\mathcal{P}}$ if and only if a machine with the same parameters is being simulated in q_1 of $\mathcal{M}_{\mathcal{F}}$ by the simulator. (This is true since when $(q_1, q_2) \in R$, then in particular the environment did send the exact same activation commands of the form (i, j, s) in q_1 and q_2.)
- If a machine with parameters (i, j, s) as above is running, the following two properties hold:
  - $\kappa_{i,j}^{s}$ in the real state q_2 is the same as the value of the corresponding simulated trace in the ideal state q_1. (This is true since these traces contain exactly the messages as received by and sent to the adversary, these are identical in q_1 and q_2 by definition of R.)
  - The same is true for the real and simulated values of δ, and for values obtained from (real or simulated) $\mathcal{G}$. (Recall that we assume that the same randomness is used in both the real and the ideal system.)

The following lemma shows that the security definition in the Bellare–Rogaway framework is exactly what we need to ensure that only "allowed" authentications happen in the (real or simulated) execution of Π; it follows directly from their definition and results.

Lemma 5.2. *In the case that* `connect`(i, j, s) *is called by the simulator, then with overwhelming probability there is a unique session id t such that $\kappa_{j,i}^{t}$ is a matching conversation for $\kappa_{i,j}^{s}$.*

Proof of Lemma 5.2. Assume that the session t does not exist or is not unique with non-negligible probability. Since the simulator $\mathcal{S}_{\text{MA}}^{\text{MS}}$ exactly simulates the experiment defined in [BR93a], it follows that $\mathcal{E}$, $\mathcal{S}_{\text{MA}}^{\text{MS}}$, and $\mathcal{A}$ then form an adversary that achieves one of the events No-Matching (if t does not exist) or Multiple-Match (if t is not unique) as defined in [BR93a] with non-negligible probability. This is a contradiction to the assumption that Π is secure by the definition of security (for No-Matching) or by [BR93a, Proposition 4.3] (for Multiple-Match). □

In order to finish the proof, we now consider every possible message that $\mathcal{E}$ or $\mathcal{A}$ can send to the protocol. Note that (except for corruption messages), there are only three different types of message from $\mathcal{E}$ or $\mathcal{A}$ that are accepted by the protocol machines.

(C.11)/(C.4) In this step, $\mathcal{E}$ sends (i, j, s) to $\mathcal{P}_{\Pi}^{\text{MS}}$ or $\mathcal{F}_{\text{MA}}^{\text{MS}}$.

In both systems, only internal operations are performed, and no message is sent to the adversary or the environment. The environment is activated next. Hence the follow-up states are again R-related.

(C.12)/(C.19) In this step, $\mathcal{A}$ sends (i, j, s, m_{in}) to $\mathcal{P}_{\Pi}^{\text{MS}}$ or $\mathcal{S}_{\text{MA}}^{\text{MS}}$.

If the (real or simulated) state of the protocol machine running with session (i, j, s) is 0, or the machine is not started, the message is ignored and the environment is activated again in both cases. In particular, the follow-up states are again R-related. Hence assume that this is not the case, then the state is greater than 0.

In both the real and the ideal system, the adversary is activated next, and obtains the message $(i, j, s, m_{\text{out}}, \delta')$, where $(m_{\text{out}}, \delta', \alpha') = \Pi(1^\eta, i, j, a, \kappa, r)$, and κ is the (real or simulated) trace of the session (i, j, s) (note that these are identical in q_1 and q_2, since $(q_1, q_2) \in R$), a is the secret information obtained from (real or simulated) $\mathcal{G}$ (and thus is identical as well in both systems, as $\mathcal{G}$ uses the same randomness by assumption), and r is the randomization used by the protocol session (i, j, s) (which again is identical in both systems). Thus the output to the adversary is identical, and the resulting follow-up states are R-related again.

(C.13)/(C.20) In this step, $\mathcal{A}$ sends (i, j, s) to $\mathcal{P}_{\Pi}^{\text{MS}}$ or $\mathcal{S}_{\text{MA}}^{\text{MS}}$.

If $\delta[i, j, s] = \texttt{false}$ or $n[i, j, s] = \texttt{true}$ in q_1, which corresponds to $state \neq 2$ in q_2, then nothing happens. Hence assume that $\delta[i, j, s] = \texttt{true}$ and $n[i, j, s] = \texttt{false}$ in q_1, and $state = 2$ in q_2.

In the real system, the reaction to the incoming message (i, j, s) is the delivery of the message (i, j, s) to the environment, which is then activated. We show that the same message is delivered to $\mathcal{E}$ in the ideal system:

Since $\delta[i, j, s] = \texttt{true}$, we know by construction of the simulator that earlier in the protocol run, $\texttt{connect}(i, j, s)$ has been called. Due to Lemma 5.2, with overwhelming probability this call determined a unique t with a matching conversation. In particular, a session (j, i, t) was started earlier (as by definition of protocols in [BR93a], protocols have at least three rounds).

In the step where $\texttt{connect}(i, j, s)$ was first performed, the message (i, j, s, t) was sent to $\mathcal{F}_{\text{MA}}^{\text{MS}}$. Hence the following messages were exchanged:

$$\begin{aligned} &1.\ \mathcal{S}_{\text{MA}}^{\text{MS}} \to \mathcal{F}_{\text{MA}}^{\text{MS}}: (i, j, s, t) \\ &2.\ \mathcal{F}_{\text{MA}}^{\text{MS}} \to \mathcal{F}_{\text{MA}}^{\text{MS}}: (j, i, t, s) \\ &3.\ \mathcal{F}_{\text{MA}}^{\text{MS}} \to \mathcal{F}_{\text{MA}}^{\text{MS}}: (i, j, s) \\ &4.\ \mathcal{F}_{\text{MA}}^{\text{MS}} \to \mathcal{S}_{\text{MA}}^{\text{MS}}: (i, j, s, t) \end{aligned} \tag{5.3}$$

and both sessions (i, j, s) and (j, s, t) have $state = 2$. Hence when $\mathcal{A}$ sends (i, j, s) to the simulator, and $\delta[i, j, s] = \texttt{true}$ as assumed above, we know that the ideal

machine running for (i, j, s) is in state 2. Upon receiving (i, j, s), the simulator forwards this message to the running ideal functionality, which forwards this tuple to the environment as required. Hence the follow-up states are R-related with overwhelming probability.

Corruption messages. It remains to show that the above remains true when the adversary $\mathcal{A}$ sends a corruption request to a (real or simulated) protocol machine.

By construction, the real system and the ideal one behave identically in this case: As soon as a machine in the real world is corrupted, it stops operating (except allowing the adversary to send and receive messages on the corresponding tapes). In the same way, as soon as the simulator receives a corruption request, it stops simulating the corresponding machine and only allows the adversary to use the tapes of the (ideal) functionality.

The only tape the adversary is given access to using the corruption mechanism is the tape shared by the protocol and the environment, hence sending messages on these tapes essentially corresponds to internal operations of the coalition of adversary and environment, in particular performing such an action in states q_1 or q_2 where $(q_1, q_2) \in R$ leads to follow-up states that are again R-related.

□

5.1.4. Secure Realizations do not Yield Secure Bellare–Rogaway Protocols

We have just shown that any mutual authentication protocol that is secure in the Bellare–Rogaway framework realizes a standard ideal functionality for mutual authentication. We note that the opposite direction does not hold directly:

Assume that a system of IITM's $\mathcal{P}_{\text{MA}}$ securely realizes one of the ideal functionalities defined above. Then we can define a system of IITM's $\mathcal{P}^*_{\text{MA}}$ that is a copy of $\mathcal{P}_{\text{MA}}$ with the following difference: 1. Before an IITM sends a message to the adversary, it appends the bit 0 to that message. 2. Any incoming message from the adversary is accepted only if the last bit is 1, but that bit is removed before processing the message. Thus, $\mathcal{P}^*_{\text{MA}}$ requires the adversary to flip each bit if it delivers a message from one IITM to another.

Obviously, the system $\mathcal{P}^*_{\text{MA}}$ securely realizes the ideal functionality if and only if $\mathcal{P}_{\text{MA}}$ securely realizes the same functionality. But this protocol cannot be secure (without modifications) in the Bellare–Rogaway framework as it explicitly needs *non*-matching conversations.

5.1.5. Extension to Authenticated Key Exchange

We note that while we focused on mutual authentication, the proof can be extended to authenticated key exchange:

Bellare and Rogaway model authenticated key exchange in [BR93a][17], and the key exchange functionality $\mathsf{F}_{2\mathrm{KE}}$ from [CH06] is a simple extension of the functionality $\mathsf{F}_{2\mathrm{MA}}$ from [CH06] referred to above.

In this way, our functionalities can be extended to the case of authenticated key exchange by including a key in certain messages. The proof carries over, with the difference that the keys distributed by the functionalities are not equal, but indistinguishable.

A similar connection has been shown by Shoup [Sho99]: Protocols for authenticated key exchange secure in the sense of Bellare–Rogaway (in Shoup's corrected version) are secure in a model defined by Shoup based on [BCK98]. The latter is in some way simulation-based, but less generic than the IITM framework or similar models:

Shoup defines a simulation-based security notion specifically for authenticated key exchange. An adversary $\mathcal{A}$ is allowed to play against a protocol in the *real world* while a transcript records certain parts of the adversary's actions. Now a protocol is secure if for each efficient adversary $\mathcal{A}$ that plays in the real-world there is another efficient adversary $\mathcal{A}^*$ that can play against an idealized authenticated key exchange protocol such that the transcript of the actions of $\mathcal{A}^*$ is computationally indistinguishable from the transcript produced by $\mathcal{A}$.

5.2. Secure Two-Round Message Exchange

In the previous section, we discussed a connection between the two frameworks used in this thesis for the case of mutual authentication. In this section we comment on the connection for the case of secure two-round message exchange.

As pointed out in Section 4.6.2, there seems to be no reasonable way to transfer the correctness definition from Chapter 3 to the IITM framework. Similarly, there seems to be no manageable way in the IITM framework to do a concrete analysis similar to the one we did in Chapter 3 as the IITM framework adds a lot of abstractions and features to the model.

5.2.1. Problems When Relating Both Models

While a connection exists for the case of mutual authentication, the situation is, unfortunately, far more complicated for secure two-round message exchange. We address some aspects that make it at least technically very tedious (and impossible without modifications to the model in Chapter 3) to show a similar connection for the case of secure two-round message exchange.

First, the cryptographic primitives used in our model for secure two-round message exchange in Chapter 3 are more complicated than in the basic case of mutual authentication in [BR93a]:

[17] As noted earlier, Shoup [Sho99] corrects a serious flaw in [BR93a], this has to be incorporated to prove the connection between both frameworks for the case of authenticated key exchange.

Bellare and Rogaway used only a long-lived key generator, which has a very simple interface (compute a bit string containing any party's private information upon request), but they left the rest to the protocol (i. e., if a protocol wants to use digital signatures, it has to implement this using only the information available through the long-lived key generator).

In contrast, our model in Chapter 3 explicitly provided a digital signature scheme; not only because this is what we used for our protocol, but also because we need to explicitly include it in the model if we want to give the adversary (partial) access to it. But if one compares the simple modeling we used in Chapter 3 (giving each party access to its private key as well as all public keys, and providing a signature oracle to the adversary) to the far more complex infrastructure used in Chapter 4 ($\mathcal{F}_{\text{SIG}}$, $\mathcal{P}_{\text{SI}}$, $\mathcal{F}_{\text{KS}^{\text{sig}}}$ in $\mathcal{P}_{\text{S2ME}}$ and $\mathcal{F}_{\text{EI}}$ in $\mathcal{F}_{\text{S2ME}}$), it is clear that one would at least have to make major modifications to the modeling in Chapter 3:

For example, client and server algorithms could include information derived from their signature keys in the messages, and upon certain assumptions (e. g., if the information is hashed and certain assumptions can be made about the hash function), such a protocol might still be considered secure; the adversary can even check if this information is correct if it corrupts a party. While such a protocol might be secure in the Bellare–Rogaway framework, it can certainly not be transferred directly to the IITM framework.

In the IITM framework, modeling corruption is a non-trivial task, especially if multiple functionalities are involved—we discussed this in Section 4.6.3.3. Thus, it is unclear if the simple corruption mechanism of Chapter 3 (providing the adversary with the signature key) is adequate if one wants to (automatically) transfer secure protocols to the IITM framework where, for example, the corruption status of an IITM instance possibly relies on the corruption status of multiple other IITM instances, and where resources may have to be shared in a non-trivial way, see comments in Section 4.6.3.3.

Similar, the addressing mechanism used in our realization $\mathcal{P}^{\text{SA}}_{\text{S2ME}}$ is not trivial, see Section 4.6.3.2, and any implementation of $\mathcal{F}_{\text{S2ME}}$ has to implement the same mechanism at least for the interface to the environment.

More importantly, the resource restrictions in the IITM framework are non-trivial. As discussed in Section 4.6.3.1, this has an influence on the design of IITM's, and it is important to keep those in mind not only before, e. g., accepting an incoming message as valid, but even before processing it (e. g., by sending it to a verification functionality).

5.2.2. Matching Conversations

In Chapter 3, our adaption of the notion of "matching conversations" from [BR93a] is not as strong as the original definition in the sense that we introduced a notion of equivalence and allowed the adversary to replace a message with an equivalent one before delivery. This could be important for the connection between the two frameworks:

For example, consider a protocol that 1. is a secure realization of the ideal two-round message exchange functionality in the IITM framework, 2. allows intermediary stations

on the network to append data to the messages sent over the network, but 3. discards these appended parts before the message is processed.

Then, the definition of matching conversations from [BR93a] would not allow to prove this protocol secure in the Bellare–Rogaway framework as each protocol session that leads both parties to accept has to have matching conversations (with overwhelming probability). But our relaxed notion of equivalent messages would allow this kind of "appended data" as long as the behavior of the receiving algorithm does not depend on it.

Note that while the above protocol is an artificial example, there exist situations where messages are altered during transfer, for example, the header information inserted into e-mails by relaying servers or the processing at SOAP intermediaries [NGM$^+$07].

6. Conclusion

In this thesis we analyzed ways to secure two-round message exchange protocols. We defined three protocols that have not been specified in detail before (but variants of which are widely used in practice). Using two different approaches, we proved the security of those protocols, taking into account common protocol elements such as timestamps and nonces, but also specifics of the setting of web services such as signed parts or different roles of servers and clients.

Although protocols like these are reckoned secure in practical applications, this work is the first that allows sound cryptographic security proofs of protocols in the two-round setting, faithfully including characteristical aspects of two-round protocols. Nevertheless, we still abstracted from many implementational details (see [BG05] for an overview of some examples), and we used the random oracle model for the analysis of our password-based protocol.

We first discussed specifics of the two-round setting as well as notions of authentication and put "message exchange authentication" in the context of message and entity authentication. Although the notion of message exchange authentication fits naturally in this context, to the best of our knowledge, it has not been studied before.

Then, we tailored the Bellare–Rogaway framework to model important specifics of the two-round protocol setting we wanted to analyze. The resulting security definition is self-contained in that understanding it does not require previous knowledge of any framework. We were then able to perform a concrete trace-based security analysis.

However, analyzing all three of our protocols in this style would have led to three different models (or a significantly more complex integrated model) for the three different security goals. In addition, the modeling of matching conversations is too strict in some situations (we already had to relax it using the notion of equivalent messages).

Simulation-based security clearly has the advantage that it leads to an easier statement of different security goals than an individual, trace-based definition, given that the reader is familiar with simulation-based security and the complex details of the IITM framework to understand all communication steps.

Moreover, the simulation-based approach allowed us to treat protocols for different tasks in a single model, as partially demonstrated by our parameterized ideal functionality. The security properties obtained by such an analysis are quite strong and hold (via composition) in an arbitrary context. The IITM framework (and related frameworks) is designed to support modular protocol analysis, which we were able to utilize for digital signatures and encryption.

However, those advantages come with a price when considering a concrete complex protocol. The formulation of both ideal functionalities and concrete implementations

in Chapter 4 is rather long and partly unintuitive (the latter are significantly more complex than their counterparts in Chapter 3). Both feature unnatural communication (bit strings to provide computing resources, status and activation messages sent to and received from the adversary and the environment), which are necessary due to how resources and activation are handled. Intuitively, one would like the environment to only access the "service" provided by the functionalities, but in the IITM framework, the environment needs to play additional roles (providing resources, checking corruption). In addition, the use of the joint-state theorem to enable realistic treatment of signatures results in a slightly different protocol from the one originally stated in Chapter 2 and from a realistic implementation.

When analyzing complex protocols like the ones in this thesis, tool support for the IITM framework would be highly desirable. While a fully automated analysis of the security of IITM's is not possible (as undecidability results exist for models that allow far less complexity), a partial analysis (for example, checking for matching tapes and "interfaces" of machines) would help designing ideal functionalities and realizations. Nevertheless, writing down functionalities (or even a simulator) is not a full proof, and a lot of subtleties only become clear when writing up proofs like in Chapter 4 or 5. But it may be possible to automate parts of the proofs using an automated theorem prover.

In our work we also discussed the relationship between a computational and a simulation-based security notion. For simple protocols, showing connections is feasible, but for more complex situations like the one in this thesis the potential results gained do not seem to justify solving all the technicalities involved.

Acknowledgment

Ich möchte mich an dieser Stelle bei einigen der Menschen bedanken, die mir diese Arbeit ermöglicht oder mich bei ihr unterstützt haben.

Zuerst gilt Dank natürlich meinem Betreuer Thomas Wilke für alle Hilfe und für seinen so wissensdurstigen und sorgfältigen Forscherdrang, aber auch für die Atmosphäre in der Arbeitsgruppe. Ebenso gilt mein Dank Henning Schnoor, der nicht nur (wie Thomas Wilke) an den publizierten Resultaten aus dieser Arbeit mitgewirkt hat, sondern, zusammen mit Detlef Kähler, mir auch immer wieder den Spaß an der Forschung vermittelt hat.

Für die Arbeit als Gutachter und die Betreuung während des Studiums, die mich mit zur Promotion motiviert haben, danke ich Ralf Küsters.

Vielen Dank natürlich auch an meine weiteren Kollegen; zuvorderst Sebastian Eggert, dessen kritischer Blick auf unsere Arbeit wichtige Einblicke ergab, sowie Max Tuengerthal für hilfreiche Diskussionen über IITMs – aber auch Dank an alle anderen, die zur so angenehmen und offenen Atmosphäre in der Arbeitsgruppe beitragen haben.

Der persönlichste Dank gilt aber natürlich meiner Frau Claudia, die mich viel mehr unterstützt hat als sie selbst glauben oder zugeben mag.

Appendix

A. The Simulator for the Trace-Based Analysis

We now define the simulator $\mathcal{S}$ for the trace-based analysis in Chapter 3.

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 $pk^{\text{sig}}_{\star}$ and a signature oracle $\Omega_{\star}$ 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
 1  let u = 0
 2  let U = newMap()
 3  let M = newMap()
 4  let 𝒜 choose a set A ⊆ IDs with |A| = n_ID
 5  choose x ≤ n_ID at random
 6  for a ∈ A
 7      let ā = userNr(a)
 8      let t_ā = 0
 9      if ā = x,
10          let pk_x^sig = pk_⋆^sig
11      else,
12          let (pk_ā^sig, sk_ā^sig) = G()
13      send (a, pk_ā^sig) to the adversary
14  simulate 𝒜
15      if 𝒜 sends Time(a, t)
16          return time(a, t)
17      if 𝒜 sends Send(p) to Cc, si
18          return clientSend(c, s, i, p)
19      if 𝒜 sends Receive(m) to Ss
20          return serverReceive(s, m)
21      if 𝒜 sends Send(p, h) to Ss
22          return serverSend(s, p, h)
23      if 𝒜 sends Receive(m) to Cc, si
24          return clientReceive(c, s, i, m)
25      if 𝒜 sends Corrupt(a) to Ω
26          return corrupt(a)
27      if 𝒜 sends Sign(a, p) to Ω
28          if D(request, p) or D(response, p) is successful
29              return ε
30          return sign(a, p)
```

$\underline{\texttt{userNr}(a)}$

31 let $\bar{a} = \texttt{lookup}(U, a)$
32 if $\bar{a} = \varepsilon$
33   let $\bar{a} = u$
34   let $u = u + 1$
35     $\texttt{add}(U, a, \bar{a})$
36 return $\bar{a}$

$\underline{\texttt{time}(a, t)}$

37 let $\bar{a} = \texttt{userNr}(a)$
38 if $t \geq t_{\bar{a}}$, set $t_{\bar{a}} = t$
39 return $t_{\bar{a}}$

$\underline{\texttt{clientSend}(c, s, i, p)}$

40 let $\bar{c} = \texttt{userNr}(c)$ and $\bar{s} = \texttt{userNr}(s)$
41 if $\mu^{i}_{\bar{c},\bar{s}} \neq \varepsilon$
42   return $(\varepsilon, 0)$
43 let r be a random l_{nonce}-bit number
44 let $\mu^{i}_{\bar{c},\bar{s}} = r$
45 let $m = E(\text{request}, c, s, r, t_{\bar{c}}, p)$
46 let $\sigma = \texttt{sign}(c, m)$
47 let $\hat{m} = E(\text{signature}, m, \sigma)$
48 return $(\hat{m}, 1)$

$\underline{\texttt{serverReceive}(s, \hat{m})}$

49 let $\bar{s} = \texttt{userNr}(s)$
50 if $\mu^{t_{\min}}_{\bar{s}} = \varepsilon$
51   let $\mu^{t_{\min}}_{\bar{s}} = t_{\bar{s}} \dotplus \text{tol}^{+}_{s}$ and $\mu^{L}_{\bar{s}} = \texttt{newMap}()$
52 try
53   let $(m, \sigma) = D(\text{signature}, \hat{m})$
54   let $(c, s', r, t, p) = D(\text{request}, m)$
55 if any error occurred while decoding
56   or $s' \neq s$ or $\texttt{verify}(c, m, \sigma) = 0$
57   or $t \leq \mu^{t_{\min}}_{\bar{s}}$ or $t > t_{\bar{s}} \dotplus \text{tol}^{+}_{s}$ or $\texttt{lookup}(\mu^{L}_{\bar{s}}, r) \neq \varepsilon$
58   return $(\varepsilon, 0, \varepsilon, \varepsilon)$
59 if $\texttt{size}(\mu^{L}_{\bar{s}}) \geq \text{cap}_{s}$
60   let $\mu^{t_{\min}}_{\bar{s}} = t_{\bar{s}} \dotplus \text{tol}^{+}_{s}$
61   for v in $\texttt{allValues}(\mu^{L}_{\bar{s}})$
62   let $(t', a) = D(\text{tuple}, v)$
63   if $t' < \mu^{t_{\min}}_{\bar{s}}$, let $\mu^{t_{\min}}_{\bar{s}} = t'$
64   for v in $\texttt{allValues}(\mu^{L}_{\bar{s}})$
65   let $(t', a) = D(\text{tuple}, v)$
66   if $t' \leq \mu^{t_{\min}}_{\bar{s}}$, $\texttt{remove}(\mu^{L}_{\bar{s}}, v)$
67 $\texttt{add}(\mu^{L}_{\bar{s}}, r, E(\text{tuple}, t, c))$
68 return $(p, 1, c, r)$

$\underline{\texttt{serverSend}(s, p, h)}$

69 let $\bar{s} = \texttt{userNr}(s)$
70 if $\mu_{\bar{s}}^{t_{\min}} = \varepsilon$
71   let $\mu_{\bar{s}}^{t_{\min}} = t_{\bar{s}} + \text{tol}_s^+$ and $\mu_{\bar{s}}^L = \texttt{newMap}()$
72 let $v = \texttt{lookup}(\mu_{\bar{s}}^L, h)$
73 if $v = \varepsilon$, return $(\varepsilon, 0, \varepsilon, \varepsilon, \mu_{\bar{s}})$
74 let $(t, c) = D(\text{tuple}, v)$
75 if $c = \varepsilon$, return $(\varepsilon, 0, \varepsilon, \varepsilon, \mu_{\bar{s}})$
76 $\texttt{remove}(\mu_{\bar{s}}^L, h)$
77 $\texttt{add}(\mu_{\bar{s}}^L, h, E(\text{tuple}, t, \varepsilon))$
78 let $m = E(\text{response}, s, c, r, p)$
79 let $\sigma = \texttt{sign}(s, m)$
80 let $\hat{m} = E(\text{signature}, m, \sigma)$
81 return $(\hat{m}, 1, c, \varepsilon)$

$\underline{\texttt{clientReceive}(c, s, i, \hat{m})}$

82 let $\bar{c} = \texttt{userNr}(c)$ and $\bar{s} = \texttt{userNr}(s)$
83 if $|\mu_{\bar{c},\bar{s}}^i| \neq l_{\text{nonce}}$, return $(\varepsilon, 0, \mu_{\bar{c},\bar{s}}^i)$
84 try
85   let $(m, \sigma) = D(\text{signature}, \hat{m})$
86   let $(s', c', r, p) = D(\text{response}, m)$
87 if any error occurred while decoding
88   or $c' \neq c$ or $s' \neq s$ or $r \neq \mu_{\bar{c},\bar{s}}^i$
89   or $\texttt{verify}(s, m, \sigma) = 0$
90   return $(\varepsilon, 0)$
91 let $\mu_{\bar{c},\bar{s}}^i = 0^{l_{\text{nonce}}+1}$
92   return $(p, 1)$

$\underline{\texttt{corrupt}(a)}$

93 let $\bar{a} = \texttt{userNr}(a)$
94 if $\bar{a} = x$
95   stop the simulation, but return no forgery
96 return $sk_{\bar{a}}^{\text{sig}}$

$\underline{\texttt{sign}(a, \beta)}$

97 let $\bar{a} = \texttt{userNr}(a)$
98 if $\bar{a} \neq x$
99   return $S(\beta, sk_{\bar{a}}^{\text{sig}})$
100 else
101   let $\sigma = \Omega_\star(\beta)$
102     $\texttt{add}(M, \beta, \sigma)$
103   return σ

$\underline{\texttt{verify}(a, \beta, \sigma)}$

104 let $\bar{a} = \texttt{userNr}(a)$
105 let $b = V(\beta, \sigma, pk_{\bar{a}}^{\text{sig}})$
106 if $\bar{a} = x$ and $b = 1$ and $\texttt{lookup}(M, \beta) = \varepsilon$
107   stop the simulation and return (β, σ) as a forgery
108 return b

B. IITM's for Secure Two-Round Message Exchange

B.1. Ideal Functionality

B.1.1. Message Exchange Functionality $\mathcal{F}_{\text{MX}}(leak, pw\text{-}auth)$

Parameters:

Description	Parameter	Type
Leakage	*leak*	$\{1\}^* \times \{0,1\}^* \to \{0,1\}^*$
Authentication Mode	*pw-auth*	$\{\texttt{true}, \texttt{false}\}$

Tapes: $\text{MX} \leftrightarrow E^{\text{c}}_{\text{MX}}$, $\text{MX} \leftrightarrow E^{\text{s}}_{\text{MX}}$, $\text{MX} \leftrightarrow \text{SM}$, $\text{MX} \dashleftarrow\dashrightarrow A_{\text{MX}}$

Variables and Initialization:

Variable	Type	Initial Value
$state, n_{\text{c}}, n_{\text{s}}$	$\mathbb{N}$	0
$p_{\text{c}}, p_{\text{s}}, pw, sid_{\text{c}}, sid_{\text{s}}, sid_A$	$\{0,1\}^*$	$\perp$
$replied_{\text{c}}, server\text{-}only, cor_{\text{c}}, cor_{\text{s}}, revealed_{\text{c}}, revealed_{\text{s}}, tested_{pw}$	$\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Request to send request message: (B.1)
`if` $(\underline{sid_{\text{c}}}, \text{Request}, \underline{p_{\text{c}}}, \underline{pw}, 1^{\underline{n_{\text{c}}}})$ is received from E^{c}_{MX} while $state = 0$, `do`
Generate η-bit nonces sid_A and sid_{s} randomly.
Send $(sid_A, \text{Request}, leak(1^\eta, p_{\text{c}}), |pw|, n_{\text{c}})$ to A_{MX} and let $state = 1$.

Approval to send request message: (B.2)
`if` $(sid_A, \text{Request}_{\text{OK}}, \underline{p'_{\text{c}}}, \underline{pw'})$ is received from A_{MX} while $state = 1$, `do`
If cor_{c},
If $p'_{\text{c}} \neq \varepsilon$, let $p_{\text{c}} = p'_{\text{c}}$.
If $pw' \neq \varepsilon$, let $pw = pw'$.
Send $(sid_{\text{s}}, \text{GetSession})$ to SM and let $state = 2$.

Resources to send request message: (B.3)
`if` $(sid_{\text{s}}, \text{Session}, 1^{\underline{n_{\text{s}}}})$ is received from SM while $state = 2$, `do`
If $|p_{\text{c}}| + |pw| \geq n_{\text{s}}$, let $state = 5$ and break.
Send $(\text{Test}_{\text{internal}}, pw)$ to SM.
Recv $(\text{Test}_{\text{internal}}, \underline{b})$ from SM.
If $\neg b$, let $state = 5$ and break.
Send $(sid_{\text{s}}, \text{Request}, p_{\text{c}})$ to E^{s}_{MX} and let $state = 3$.

Request to send response message: (B.4)
`if` $(sid_{\text{s}}, \text{Response}, \underline{p_{\text{s}}})$ is received from E^{s}_{MX} while $state = 3$, `do`
If $server\text{-}only \vee replied_{\text{c}}$, let $state = 5$, else let $state = 4$.
Send $(sid_A, \text{Response}, leak(1^\eta, p_{\text{s}}))$ to A_{MX}.

Approval to send response message: (B.5)
`if` $(sid_A, \text{Response}_{\text{OK}}, \underline{p'_{\text{s}}})$ is received from A_{MX} while $(state = 4) \vee (cor_{\text{s}} \wedge (state \in \{1,2,3\}))$, `do`
If $(p'_{\text{s}} \neq \varepsilon) \wedge cor_{\text{s}}$ let $p_{\text{s}} = p'_{\text{s}}$.
If $|p_{\text{s}}| \geq n_{\text{c}}$, let $state = 5$ and break.
If $state = 4$ then let $state = 5$, else let $replied_{\text{c}} = \texttt{true}$.
Send $(sid_{\text{c}}, \text{Response}, p_{\text{s}})$ to E^{c}_{MX}.

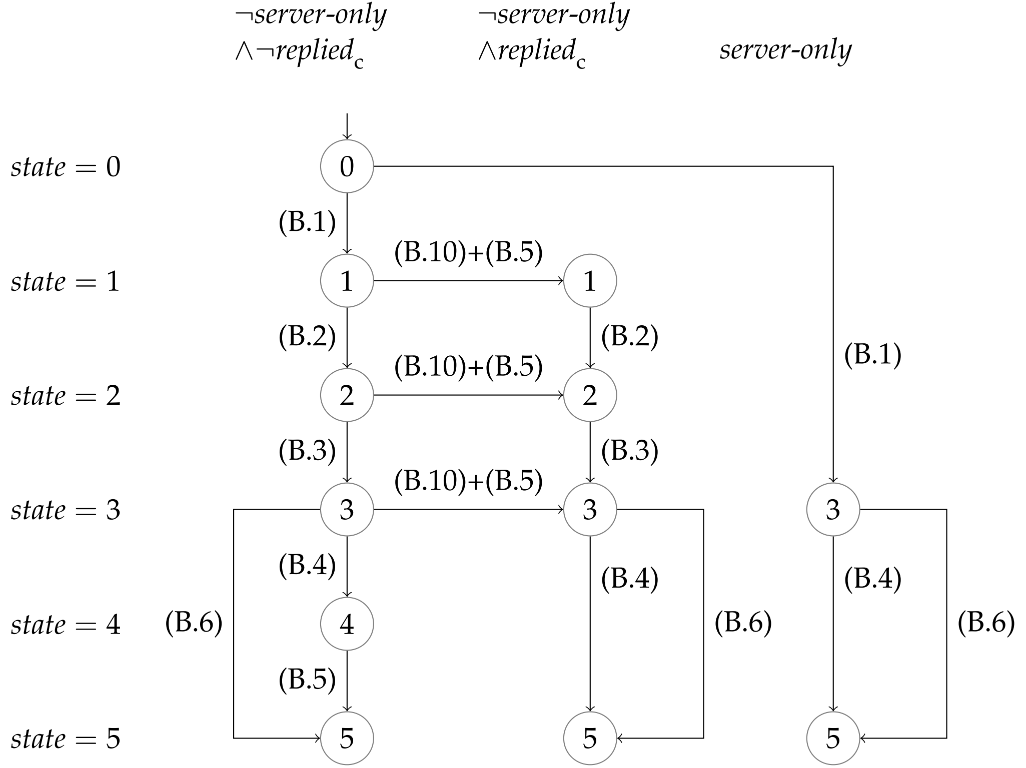

Figure B.1.: States and steps of the functionality $\mathcal{F}_{\mathrm{MX}}$

Expire this session on server side: (B.6)
`if` (sid_A, Expire) is received from A_{MX} while $state = 3$, `do`
  Send $(sid_A, \text{Expire}_{\mathrm{OK}})$ to A_{MX} and let $state = 5$.

Request to send response message in non-existent or expired session: (B.7)
`if` $(\underline{sid_{\mathrm{s}}}, \text{Response}, \underline{p_{\mathrm{s}}})$ received from $E^{\mathrm{s}}_{\mathrm{MX}}$ with $sid_s \neq \varepsilon$ while $state = 0$, `or`
`if` $(sid_{\mathrm{s}}, \text{Response}, \underline{p_{\mathrm{s}}})$ received from $E^{\mathrm{s}}_{\mathrm{MX}}$ while $state > 3$, `do`
  Send $(sid_{\mathrm{s}}, \text{Response}_{\mathrm{Error}})$ to $E^{\mathrm{s}}_{\mathrm{MX}}$ and if $state = 0$, halt.

Start of a server-only session: (B.8)
`if` $(\text{Session}, \underline{cor_{\mathrm{c}}}, \underline{p_{\mathrm{c}}})$ is received from SM while $state = 0$, `do`
  Generate η-bit nonces sid_A and sid_{s} randomly.
  Send $(sid_A, \text{Session})$ to A_{MX}.
  Recv $(sid_A, \text{Session}_{\mathrm{OK}})$ from A_{MX}.
  Send $(sid_{\mathrm{s}}, \text{Request}, p_{\mathrm{c}})$ to $E^{\mathrm{s}}_{\mathrm{MX}}$, let $server\text{-}only = \texttt{true}$ and let $state = 3$.

Test if password is correct: (B.9)
`if` (sid_A, Test) is received from A_{MX} while $(state = 1) \wedge pw\text{-}auth \wedge \neg tested_{pw}$, `do`
  Send (Test, pw) to SM and let $tested_{pw} = \texttt{true}$.

Corrupt this session: (B.10)
`if` $(sid_A, \text{Corrupt}, \underline{x})$ is received from A_{MX} with $x \in \{\mathsf{c}, \mathsf{s}\}$ while $(state > 0) \wedge \neg cor_x$, `do`
  Send $(sid_A, \text{Corrupt}_{\mathrm{OK}}, x)$ to A_{MX} and let $cor_x = \texttt{true}$.

Reveal payload and password in a corrupted session: (B.11)
`if` $(sid_A, \text{Reveal}, \underline{x})$ is received from A_{MX} with $x \in \{\mathsf{c}, \mathsf{s}\}$ while $(state > 0) \wedge cor_x \wedge \neg revealed_x$, `do`
  Send $(sid_A, \text{Reveal}, x, p_x, pw)$ to A_{MX} and let $revealed_x = \texttt{true}$.

Corruption status: (B.12)
`if` $(sid_x, \text{Corrupted?})$ is received from E^x_{MX} with $((x = \text{c}) \wedge (state > 0) \wedge \neg server\text{-}only) \vee ((x = \text{s}) \wedge (state > 2))$, `do`
Send $(sid_x, \text{Corrupted}, cor_x)$ to E^x_{MX}.

Provide resources for corrupted sessions: (B.13)
`if` $(sid_x, \text{Resources}, 1^{\underline{n'}})$ is received from E^x_{MX} with $x \in \{\text{c}, \text{s}\}$ while $state > 0$, `do`
Send $(sid_A, \text{Resources}, x, 1^{n'})$ to A_{MX}.

CheckAddress:

- If $state = 0$, accept any message.
- If $sid_\text{c} \neq \bot$, accept all messages from E^c_{MX} starting with sid_c.
- If $sid_\text{s} \neq \bot$, accept all messages from E^s_{MX} and SM starting with sid_s.
- If $sid_A \neq \bot$, accept all messages from A_{MX} starting with sid_A.

B.1.2. Server Management Functionality $\mathcal{F}_{\text{SM}}(pw\text{-}auth)$

Parameters:

Description	Parameter	Type
Authentication Mode	*pw-auth*	{`true`, `false`}

Tapes: $\text{SM} \twoheadleftarrow\!\!\longrightarrow E_{\text{SM}}$, $\text{SM} \longleftarrow\!\!\twoheadrightarrow \text{MX}$, $\text{SM} \dashleftarrow\!\dashrightarrow A_{\text{SM}}$

Variables and Initialization:

Variable	Type	Initial Value
state	$\mathbb{N}$	0
U	$\{0,1\}^* \to \{0,1\}^*$	$\bot$
n	$\mathbb{N}$	0

Steps: `loop`

Initialization and Users: (B.14)
`if` $(\text{Init}, \underline{U})$ is received from E_{SM} while $state = 0$, `do`
Send (Init) to A_{SM}.
Recv $(\text{Init}_{\text{OK}})$ from A_{SM}.
Set $state = 1$.

Receive resources: (B.15)
`if` $(\text{Resources}, 1^{\underline{n}})$ is received from E_{SM} while $state = 1$, `do`
Send $(\text{Resources}, 1^n)$ to A_{SM}.

Initialize a regular session: (B.16)
`if` $(\underline{c}, \underline{sid_\text{s}}, \text{GetSession})$ is received from MX while $(state = 1) \wedge (n > 0)$, `do`
Send $(c, sid_\text{s}, \text{Session}, 1^n)$ to MX and let $n = 0$.

Initialize a server-only session with the correct password: (B.17)
`if` $(\text{Session}, \underline{c}, \underline{cor}, \underline{pw}, \underline{p_\text{c}})$ is received from A_{SM} while $(state = 1) \wedge pw\text{-}auth$, `do`
If $|p_\text{c}| + |pw| \geq n$, break.
Let $n = 0$.
If $U(c) \neq pw$, break.
Send $(c, \text{Session}, cor, p_\text{c})$ to MX.

Let the adversary test a password: (B.18)
`if` $(\underline{c}, \text{Test}, \underline{pw})$ is received from A_{SM} or MX while $pw\text{-}auth \wedge (n > 0)$, `do`
Let $n = n - 1$.
Send $(c, \text{Test}, U(c) = pw)$ to A_{SM}.

Internally test a password: (B.19)
`if` $(\underline{c}, \text{Test}_{\text{internal}}, \underline{pw})$ is received from MX while $pw\text{-}auth$, `do`
Send $(c, \text{Test}_{\text{internal}}, U(c) = pw)$ to MX.

B.1.3. Enriching Input Functionality $\mathcal{F}_{\text{EI}}$

Tapes: $\text{EI} \twoheadleftarrow E_{\text{EI}}, \text{EI} \dashrightarrow A_{\text{EI}}$

Steps: `loop`

Forward resources: (B.20)
`if` $(\text{Resources}, 1^{\underline{n}})$ is received from E_{EI}, `do`
  Send $(\text{Resources}, n)$ to A_{EI}.

B.2. Realization

B.2.1. Client Functionality (SA) $\mathcal{P}_{\text{C}}^{\text{SA}}$

Tapes: $\text{C} \twoheadleftarrow\!\!\rightarrow E_{\text{MX}}^{\text{C}}, \text{C} \dashleftarrow\!\!\dashrightarrow A_{\text{C}}, \text{C} \leftarrow\!\!\twoheadrightarrow \text{KS}^{\text{sig}}, \text{C} \leftarrow\!\!\twoheadrightarrow \text{LC}, \text{C}_{\text{sig}} \leftarrow\!\!\twoheadrightarrow \text{SIG}, \text{C}_{\text{ver}} \leftarrow\!\!\twoheadrightarrow \text{SIG}$

Variables and Initialization:

Variable	Type	Initial Value
$state, n_c$	$\mathbb{N}$	0
s, c, sid_c, r	$\{0,1\}^*$	$\perp$

Steps: `loop`

Send a request to the server: (B.21)
`if` $(\underline{s}, \underline{c}, \underline{sid_{\text{c}}}, \text{Request}, \underline{p_{\text{c}}}, \underline{pw}, 1^{\underline{n_{\text{c}}}})$ is received from E_{MX}^{c} while $state = 0$, `do`
  Let $state = 1$.
  Generate an η-bit nonce r randomly.
  Send $(c, (\text{C}, s, r), \text{GetTime})$ to LC.
  Recv $(c, (\text{C}, s, r), \text{Time}, \underline{t})$ from LC.
  Let $m_{\text{c}} = (\text{From}: c, \text{To}: s, \text{MsgID}: r, \text{Time}: t, \text{Body}: p_{\text{c}})$.
  Send $(c, (\text{C}, s, r), \text{GetKey})$ to KS^{sig}.
  Recv $(c, (\text{C}, s, r), \text{PublicKey}, \underline{pk_{\text{c}}^{\text{sig}}})$ from KS^{sig}.
  Send $(c, (\text{C}, s, r), \text{Sign}, m_{\text{c}})$ to SIG on C_{sig}.
  Recv $(c, (\text{C}, s, r), \text{Signature}, \underline{\sigma_{\text{c}}})$ from SIG on C_{sig}.
  Send $(m_{\text{c}}, \sigma_{\text{c}})$ to A_{C} and let $state = 1$.

Receive and process a response from the server: (B.22)
`if` $(\underline{m_{\text{s}}}, \underline{\sigma_{\text{s}}})$ is received from A_{C} with $m_{\text{s}} = (\text{From}: c, \text{To}: s, \text{Ref}: r, \text{Body}: \underline{p_{\text{s}}})$ while $state = 1$, `do`
  Let $state = 3$.
  If $|p_{\text{s}}| \geq n_{\text{c}}$, break.
  Send $(s, (\text{S}, c, r), \text{GetKey})$ to KS^{sig}.
  Recv $(s, (\text{S}, c, r), \text{PublicKey}, \underline{pk_{\text{s}}^{\text{sig}}})$ from KS^{sig}.
  Send $(s, (\text{S}, c, r), \text{C}, \text{Init})$ to SIG on C_{ver}.
  Recv $(s, (\text{S}, c, r), \text{C}, \text{Inited})$ from SIG on C_{ver}.
  Send $(s, (\text{S}, c, r), \text{C}, \text{Verify}, m_{\text{s}}, \sigma_{\text{s}}, pk_{\text{s}}^{\text{sig}})$ to SIG on C_{ver}.
  Recv $(s, (\text{S}, c, r), \text{C}, \text{Verified}, \underline{b})$ from SIG on C_{ver}.
  If $\neg b$, break.
  Send $(s, c, sid_{\text{c}}, \text{Response}, p_{\text{s}})$ to E_{MX}^{c}.

Provide resources for corrupted signature scheme: (B.23)
`if` $(s, c, sid_{\text{c}}, \text{Corrupt}, 1^{\underline{n'}})$ is received from E_{MX}^{c} while $state > 0$, `do`
  Send $(c, (\text{C}, s, r), \text{Resources}, 1^{n'})$ to KS^{sig}.

Corruption status: (B.24)
`if` $(s, c, sid_{\text{c}}, \text{Corrupted?})$ is received from E_{MX}^{c} while $state > 0$, `do`
  Send $(c, (\text{C}, s, r), \text{Corrupted?})$ to KS^{sig}.
  Recv $(c, (\text{C}, s, r), \text{Corrupted}, \underline{cor_1})$ from KS^{sig}.
  Send $(c, (\text{C}, s, r), \text{S}, \text{Corrupted?})$ to KS^{sig}.
  Recv $(c, (\text{C}, s, r), \text{S}, \text{Corrupted}, \underline{cor_2})$ from KS^{sig}.
  Send $(s, c, sid_{\text{c}}, \text{Corrupted}, cor_1 \vee cor_2)$ to E_{MX}^{c}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.2. Client Functionality (CSA) $\mathcal{P}_{\mathrm{C}}^{\mathrm{CSA}}$

Tapes: $\mathrm{C} \twoheadleftarrow\!\!\longrightarrow E_{\mathrm{MX}}^{\mathrm{C}}$, $\mathrm{C} \dashleftarrow\dashrightarrow A_{\mathrm{C}}$, $\mathrm{C} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{KS}^{\mathrm{ae}}$, $\mathrm{C} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{KS}^{\mathrm{sig}}$, $\mathrm{C} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{LC}$, $\mathrm{C} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{ENC}$, $\mathrm{C}_{\mathrm{sig}} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{SIG}$, $\mathrm{C}_{\mathrm{ver}} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{SIG}$

Variables and Initialization:

Variable	Type	Initial Value
$state, n_c$	$\mathbb{N}$	0
s, c, sid_c, r	$\{0,1\}^*$	$\perp$
ptr	key pointer	$\perp$

Steps: `loop`

Send a request to the server: (B.25)

`if` $(\underline{s}, \underline{c}, \underline{sid_c}, \text{Request}, \underline{p_c}, \underline{pw}, 1^{\underline{n_c}})$ is received from $E_{\mathrm{MX}}^{\mathrm{c}}$ while $state = 0$, `do`

Let $state = 1$.
Generate an η-bit nonce r randomly.
Send $(c, (\mathrm{C}, s, r), \text{GetTime})$ to LC.
Recv $(c, (\mathrm{C}, s, r), \text{Time}, \underline{t})$ from LC.
Send $((s, c, r), \text{KeyGen})$ to ENC.
Recv $((s, c, r), \text{KeyGen}, \underline{ptr})$ from ENC.
Send $(s, (\mathrm{C}, c, r), \text{GetKey})$ to $\mathrm{KS}^{\mathrm{ae}}$.
Recv $(s, (\mathrm{C}, c, r), \text{PublicKey}, \underline{pk^{\mathrm{ae}}})$ from $\mathrm{KS}^{\mathrm{ae}}$.
Send $(s, (\mathrm{C}, c, r), \text{Initialize})$ to ENC.
Recv $(s, (\mathrm{C}, c, r), \text{Completed})$ from ENC.
Send $(s, (\mathrm{C}, c, r), \text{Enc}, pk^{\mathrm{ae}}, (\text{Key}, ptr))$ to ENC.
Recv $(s, (\mathrm{C}, c, r), \text{Ciphertext}, \underline{\varrho_k})$ from ENC.
Send $((s, c, r), \text{Enc}, ptr, p_c)$ to ENC.
Recv $((s, c, r), \text{Ciphertext}, \underline{\varrho_c})$ from ENC.
Let $m_c = (\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon r, \text{Time}\colon t, \text{Key}\colon \varrho_k, \text{Body}\colon \varrho_c)$.
Send $(c, (\mathrm{C}, s, r), \text{GetKey})$ to $\mathrm{KS}^{\mathrm{sig}}$.
Recv $(c, (\mathrm{C}, s, r), \text{PublicKey}, \underline{pk_c^{\mathrm{sig}}})$ from $\mathrm{KS}^{\mathrm{sig}}$.
Send $(c, (\mathrm{C}, s, r), \text{Sign}, m_c)$ to SIG on $\mathrm{C}_{\mathrm{sig}}$.
Recv $(c, (\mathrm{C}, s, r), \text{Signature}, \underline{\sigma_c})$ from SIG on $\mathrm{C}_{\mathrm{sig}}$.
Send (m_c, σ_c) to A_{C} and let $state = 1$.

Receive and process a response from the server: (B.26)

`if` $(\underline{m_s}, \underline{\sigma_s})$ is received from A_{C} with $m_s = (\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon r, \text{Body}\colon \underline{\varrho_s})$ while $state = 1$, `do`

Let $state = 3$.
If $|\varrho_s| \geq n_c$, break.
Send $(s, (\mathrm{S}, c, r), \text{GetKey})$ to $\mathrm{KS}^{\mathrm{sig}}$.
Recv $(s, (\mathrm{S}, c, r), \text{PublicKey}, \underline{pk_s^{\mathrm{sig}}})$ from $\mathrm{KS}^{\mathrm{sig}}$.
Send $(s, (\mathrm{S}, c, r), \mathrm{C}, \text{Init})$ to SIG on $\mathrm{C}_{\mathrm{ver}}$.
Recv $(s, (\mathrm{S}, c, r), \mathrm{C}, \text{Inited})$ from SIG on $\mathrm{C}_{\mathrm{ver}}$.
Send $(s, (\mathrm{S}, c, r), \mathrm{C}, \text{Verify}, m_s, \sigma_s, pk_s^{\mathrm{sig}})$ to SIG on $\mathrm{C}_{\mathrm{ver}}$.
Recv $(s, (\mathrm{S}, c, r), \mathrm{C}, \text{Verified}, \underline{b})$ from SIG on $\mathrm{C}_{\mathrm{ver}}$.
If $\neg b$, break.
Send $((s, c, r), \text{Dec}, ptr, \varrho_s)$ to ENC.
Recv $((s, c, r), \text{Plaintext}, \underline{p_s})$ from ENC.
Send $(s, c, sid_c, \text{Response}, p_s)$ to $E_{\mathrm{MX}}^{\mathrm{c}}$.

Provide resources for corrupted signature scheme: (B.27)

`if` $(s, c, sid_c, \text{Corrupt}, 1^{\underline{n'}})$ is received from $E_{\mathrm{MX}}^{\mathrm{c}}$ while $state > 0$, `do`

Send $(c, (\mathrm{C}, s, r), \text{Resources}, 1^{n'})$ to $\mathrm{KS}^{\mathrm{sig}}$.

Corruption status: (B.28)
`if` $(s, c, sid_c, \text{Corrupted?})$ is received from E^c_{MX} while $state > 0$, `do`
Send $(s, (C, c, r), \text{Corrupted?})$ to KS^{ae}.
Recv $(s, (C, c, r), \text{Corrupted}, \underline{cor_1})$ from KS^{ae}.
Send $(c, (C, s, r), \text{Corrupted?})$ to KS^{sig}.
Recv $(c, (C, s, r), \text{Corrupted}, \underline{cor_2})$ from KS^{sig}.
Send $(c, (C, s, r), S, \text{Corrupted?})$ to KS^{sig}.
Recv $(c, (C, s, r), S, \text{Corrupted}, \underline{cor_3})$ from KS^{sig}.
Send $((s, c, r), \text{Corrupted?}, ptr)$ to ENC.
Recv $((s, c, r), \text{CorruptionState}, \underline{cor_4})$ from ENC.
Send $(s, c, sid_c, \text{Corrupted}, cor_1 \vee cor_2 \vee cor_3 \vee cor_4)$ to E^c_{MX}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.3. Client Functionality (PA) $\mathcal{P}^{PA}_{C}$

Tapes: $C \twoheadleftarrow\!\!\longrightarrow E^C_{MX}$, $C \dashleftarrow\!\dashrightarrow A_C$, $C \longleftrightarrow\!\!\!\twoheadrightarrow KS^{ae}$, $C \longleftrightarrow\!\!\!\twoheadrightarrow KS^{sig}$, $C \longleftrightarrow\!\!\!\twoheadrightarrow LC$, $C \longleftrightarrow\!\!\!\twoheadrightarrow ENC$, $C_{sig} \longleftrightarrow\!\!\!\twoheadrightarrow SIG$, $C_{ver} \longleftrightarrow\!\!\!\twoheadrightarrow SIG$

Variables and Initialization:

Variable	Type	Initial Value
$state, n_c$	$\mathbb{N}$	0
s, c, sid_c, r, H_r	$\{0,1\}^*$	$\perp$

Steps: `loop`

Send a request to the server: (B.29)
`if` $(\underline{s}, \underline{c}, \underline{sid_c}, \text{Request}, \underline{p_c}, \underline{pw}, 1^{\underline{n_c}})$ is received from E^c_{MX} while $state = 0$, `do`
Let $state = 1$.
Generate an η-bit nonce r_1 randomly.
Send (GetRO, r) to RO.
Recv $(\text{RO}, \underline{H_r})$ from RO.
Send $(c, (C, s, H_r), \text{GetTime})$ to LC.
Recv $(c, (C, s, H_r), \text{Time}, \underline{t})$ from LC.
Let $m_c = (\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon H_r, \text{Time}\colon t, \text{Body}\colon p_c)$.
Send (GetRO, m_c) to RO.
Recv $(\text{RO}, \underline{H_{m_c}})$ from RO.
Let $m'_c = (\text{SecMsgID}\colon r, \text{Pass}\colon pw, \text{MsgHash}\colon H_{m_c})$.
Send $(s, (C, c, H_r), \text{GetKey})$ to KS^{ae}.
Recv $(s, (C, c, H_r), \text{PublicKey}, \underline{pk^{ae}})$ from KS^{ae}.
Send $(s, (C, c, H_r), \text{Initialize})$ to ENC.
Recv $(s, (C, c, H_r), \text{Completed})$ from ENC.
Send $(s, (C, c, H_r), \text{Enc}, pk^{ae}, m'_c)$ to ENC.
Recv $(s, (C, c, H_r), \text{Ciphertext}, \underline{\varrho_c})$ from ENC.
Send (m_c, ϱ_c) to A_C and let $state = 1$.

Receive and process a response from the server: (B.30)
`if` $(\underline{m_s}, \underline{\sigma_s})$ is received from A_C with $m_s = (\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon H_r, \text{Body}\colon \underline{p_s})$ while $state = 1$, `do`
Let $state = 3$.
If $|p_s| \geq n_c$, break.
Send $(s, (S, c, H_r), \text{GetKey})$ to KS^{sig}.
Recv $(s, (S, c, H_r), \text{PublicKey}, \underline{pk^{sig}_s})$ from KS^{sig}.
Send $(s, (S, c, H_r), C, \text{Init})$ to SIG on C_{ver}.
Recv $(s, (S, c, H_r), C, \text{Inited})$ from SIG on C_{ver}.
Send $(s, (S, c, H_r), C, \text{Verify}, m_s, \sigma_s, pk^{sig}_s)$ to SIG on C_{ver}.
Recv $(s, (S, c, H_r), C, \text{Verified}, \underline{b})$ from SIG on C_{ver}.
If $\neg b$, break.
Send $(s, c, sid_c, \text{Response}, p_s)$ to E^c_{MX}.

Provide resources for corruption: (B.31)
`if` $(s, c, sid_c, \text{Corrupt}, 1^{\underline{n'}})$ is received from E^c_{MX} while $state > 0$, `do`
Break.

Corruption status: (B.32)
`if` $(s, c, sid_c, \text{Corrupted?})$ is received from E^c_{MX} while $state > 0$, `do`
Send $(s, (C, c, H_r), \text{Corrupted?})$ to KS^{ae}.
Recv $(s, (C, c, H_r), \text{Corrupted}, \underline{cor})$ from KS^{ae}.
Send $(s, c, sid_c, \text{Corrupted}, cor)$ to E^c_{MX}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.4. Server Functionality (SA) $\mathcal{P}^{SA}_S$

Tapes: $S \twoheadleftarrow\!\!\longrightarrow E^S_{MX}$, $S \dashleftarrow\!\!\dashrightarrow A_S$, $S \longleftrightarrow\!\!\!\!\twoheadrightarrow KS^{sig}$, $S \longleftrightarrow\!\!\!\!\twoheadrightarrow LC$, $S_{sig} \longleftrightarrow\!\!\!\!\twoheadrightarrow SIG$, $S_{ver} \longleftrightarrow\!\!\!\!\twoheadrightarrow SIG$

Variables and Initialization:

Variable	Type	Initial Value
$state, n, \text{cap}$	$\mathbb{N}$	0
U	$\{0,1\}^* \to \{0,1\}^*$	$\perp$
$c, s, r, m_c, m_s, \sigma_c, p_c, p_s, pk^{sig}_c, pk^{sig}_s, t_c, t_s, t_{min}, \text{tol}^+$	$\{0,1\}^*$	$\perp$
L, L_{cor}	sets of 4-tuples of $\{0,1\}^*$	[ ]

Steps: `loop`

Initialization and Users: (B.33)
`if` $(\underline{s}, \text{Init}, \underline{U})$ is received from E_{SM} while $state = 0$, `do`
Send $(s, \text{GetParameters})$ to A_S.
Recv $(s, \text{Parameters}, \underline{\text{cap}}, \underline{\text{tol}^+})$ from A_S with $\text{cap} > 0$ and $\text{tol}^+ > 0$.
Send $(s, S, \text{GetTime})$ to LC.
Recv $(s, S, \text{Time}, \underline{t_s})$ from LC.
Let $t_{min} = t_s + \text{tol}^+$ and let $state = 1$.

Receive resources: (B.34)
`if` $(s, \text{Resources}, 1^{\underline{n}})$ is received from E_{SM} while $state > 0$, `do`
Break.

Receive and process a request: Request the client's key: (B.35)
`if` $(\underline{m_c}, \underline{\sigma_c})$ is received from A_S with $m_c = (\text{From}: \underline{c}, \text{To}: s, \text{MsgID}: \underline{r}, \text{Time}: \underline{t_c}, \text{Body}: \underline{p_c})$ while $state > 0$, `do`
If $|p_c| \geq n$, break.
Let $n = 0$.
Send $(c, (C, s, r), \text{GetKey})$ to KS^{sig} and let $state = 2$.

Receive and process a request: Receive the key, request time: (B.36)
`if` $(c, (C, s, r), \text{PublicKey}, \underline{pk^{sig}_c})$ is received from KS^{sig} while $state = 2$, `do`
Send $(s, S, \text{GetTime})$ to LC and let $state = 3$.

Receive and process a request: Receive time, initialize the verifier: (B.37)
`if` $(s, S, \text{Time}, \underline{t_s})$ is received from LC while $state = 3$, `do`
Send $(c, (C, s, r), S, \text{Init})$ on S_{ver} and let $state = 4$.

Receive and process a request: Verifier initialized, request verification: (B.38)
`if` $(c, (C, s, r), S, \text{Inited})$ is received on S_{ver} while $state = 4$, `do`
Send $(c, (C, s, r), S, \text{Verify}, m_c, \sigma_c, pk^{sig}_c)$ on S_{ver} and let $state = 5$.

Receive and process a request: Execute protocol steps, relay request: (B.39)
`if` $(c, (C, s, r), S, \text{Verified}, \underline{b})$ is received on S_{ver} while $state = 5$, `do`
If $(\neg b) \vee (t_c \leq t_{min}) \vee (t_c > t_s + \text{tol}^+) \vee (\exists t', c', sid'_s : (t', r, c', sid'_s) \in L)$, break.
While $|L| \geq \text{cap}$:
Let $t_{min} = \min\{t' \mid (t', r', c', sid'_s) \in L\}$ and $L = \{(t', r', c', sid'_s) \in L \mid t' > t_{min}\}$.
Generate an η-bit nonce sid_s randomly.
Insert (t_c, r, c, sid_s) into L and L_{cor}.
Send $(s, c, sid_s, \text{Request}, p_c)$ to E^s_{MX} and let $state = 1$.

Receive and process a response: Receive response payload, request key: (B.40)
`if` $(s, c, \underline{sid_s}, \text{Response}, \underline{p_s})$ is received from E^s_{MX} with $sid_s \neq \varepsilon$ while $state > 0$, `do`
    If $\neg\exists t, r, c\colon (t, r, c, sid_s) \in L$,
        Send $(s, sid_s, \text{Response}_{\text{Error}})$ to E^s_{MX} and break.
    Fetch $(\underline{t_c}, \underline{r}, \underline{c}, sid_s)$ from L.
    Update the entry (t_c, r, c, sid_s) in L to (t_c, r, c, ε).
    Let $m_s = (\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon r, \text{Body}\colon p_s)$.
    Send $(s, (S, c, r), \text{GetKey})$ to KS^{sig} and let $state = 6$.

Receive and process a response: Construct response message and request signature: (B.41)
`if` $(s, (S, c, r), \text{PublicKey}, \underline{pk^{\text{sig}}_s})$ is received from KS^{sig} while $state = 6$, `do`
    Send $(s, (S, c, r), \text{Sign}, m_s)$ on S_{sig} and let $state = 7$.

Receive and process a response: Receive signature, send out message: (B.42)
`if` $(s, (S, c, r), \text{Signature}, \underline{\sigma_s})$ is received on S_{sig} while $state = 7$, `do`
    Send (m_s, σ_s) to A_S and let $state = 1$.

Reset the server: (B.43)
`if` (s, Reset) is received from A_S while $state > 0$, `do`
    Let $t_{\min} = t_s + \text{tol}^+$, let $L = [\,]$ and $state = 1$.

Request to send response message for non-initialized server: (B.44)
`if` $(\underline{s}, \underline{c}, \underline{sid_s}, \text{Response}, \underline{p_s})$ is received from E^s_{MX} with $sid_s \neq \varepsilon$ while $state = 0$, `do`
    Send $(s, sid_s, \text{Response}_{\text{Error}})$ to E^s_{MX} and halt.

Provide resources for corrupted signature scheme: (B.45)
`if` $(s, \underline{sid'_s}, \text{Corrupt}, 1^{\underline{n'}})$ is received from E^s_{MX} while $state > 0$, `do`
    If $\neg\exists t', r', c'\colon (t', r', c', sid'_s) \in L_{cor}$, break.
    Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, sid'_s)$ from L_{cor}.
    Send $(s, (S, c', r'), \text{Resources}, 1^{n'})$ to KS^{sig}.

Corruption status: (B.46)
`if` $(s, sid_s, \text{Corrupted?})$ is received from E^s_{MX} while $state > 0$, `do`
    If $\neg\exists t', r', c'\colon (t', r', c', sid_s) \in L_{cor}$, break.
    Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, sid_s)$ from L_{cor}.
    Send $(s, (S, c', r'), \text{Corrupted?})$ to KS^{sig}.
    Recv $(s, (S, c', r'), \text{Corrupted}, \underline{cor_1})$ from KS^{sig}.
    Send $(s, (S, c', r'), C, \text{Corrupted?})$ to KS^{sig}.
    Recv $(s, (S, c', r'), C, \text{Corrupted}, \underline{cor_2})$ from KS^{sig}.
    Send $(s, sid_s, \text{Corrupted}, cor_1 \vee cor_2)$ to E^s_{MX}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.5. Server Functionality (CSA) $\mathcal{P}^{\text{CSA}}_{S}$

Tapes: $S \twoheadleftarrow\!\!\rightarrow E^S_{MX}$, $S \leftarrow\!\!\dashrightarrow A_S$, $S \longleftrightarrow\!\!\!\!\!\rightarrow \text{KS}^{\text{ae}}$, $S \longleftrightarrow\!\!\!\!\!\rightarrow \text{KS}^{\text{sig}}$, $S \longleftrightarrow\!\!\!\!\!\rightarrow \text{LC}$, $S \longleftrightarrow\!\!\!\!\!\rightarrow \text{ENC}$, $S_{\text{sig}} \longleftrightarrow\!\!\!\!\!\rightarrow \text{SIG}$, $S_{\text{ver}} \longleftrightarrow\!\!\!\!\!\rightarrow \text{SIG}$

Variables and Initialization:

Variable	Type	Initial Value
$state, n, \text{cap}$	$\mathbb{N}$	0
U	$\{0,1\}^* \to \{0,1\}^*$	$\perp$
$c, s, r, m_c, m_s, \sigma_c, \varrho_k, \varrho_c, p_c, p_s, pk^{\text{sig}}_c, pk^{\text{sig}}_s, t_c, t_s, t_{\min}, \text{tol}^+, b$	$\{0,1\}^*$	$\perp$
L, L_{cor}	sets of 5-tuples of $\{0,1\}^*$	$[\,]$

Steps: `loop`

Initialization and Users: (B.47)
`if` $(\underline{s}, \mathsf{Init}, \underline{U})$ is received from E_{SM} while $state = 0$, `do`
Send $(s, \mathsf{GetParameters})$ to A_{S}.
Recv $(s, \mathsf{Parameters}, \underline{\mathsf{cap}}, \underline{\mathsf{tol}^+})$ from A_{S} with cap > 0 and $\mathsf{tol}^+ > 0$.
Send $(s, \mathsf{S}, \mathsf{GetKey})$ to $\mathsf{KS}^{\mathsf{ae}}$.
Recv $(s, \mathsf{S}, \mathsf{PublicKey}, \underline{pk^{\mathsf{ae}}})$ from $\mathsf{KS}^{\mathsf{ae}}$.
Send $(s, \mathsf{S}, \mathsf{GetTime})$ to LC.
Recv $(s, \mathsf{S}, \mathsf{Time}, \underline{t_{\mathrm{s}}})$ from LC.
Let $t_{\min} = t_{\mathrm{s}} + \mathsf{tol}^+$ and let $state = 1$.

Receive resources: (B.48)
`if` $(s, \mathsf{Resources}, 1^{\underline{n}})$ is received from E_{SM} while $state > 0$, `do`
Break.

Receive and process a request: Request the client's key: (B.49)
`if` $(\underline{m_{\mathrm{c}}}, \underline{\sigma_{\mathrm{c}}})$ is received from A_{S} with $m_{\mathrm{c}} = (\mathsf{From}\colon \underline{c}, \mathsf{To}\colon s, \mathsf{MsgID}\colon \underline{r}, \mathsf{Time}\colon \underline{t_{\mathrm{c}}}, \mathsf{Key}\colon \underline{\varrho_k}, \mathsf{Body}\colon \underline{\varrho_{\mathrm{c}}})$ while $state > 0$, `do`
If $|\varrho_{\mathrm{c}}| \geq n$, break.
Let $n = 0$.
Send $(c, (\mathsf{C}, s, r), \mathsf{GetKey})$ to $\mathsf{KS}^{\mathsf{sig}}$ and let $state = 2$.

Receive and process a request: Receive the key, request time: (B.50)
`if` $(c, (\mathsf{C}, s, r), \mathsf{PublicKey}, \underline{pk_{\mathrm{c}}^{\mathsf{sig}}})$ is received from $\mathsf{KS}^{\mathsf{sig}}$ while $state = 2$, `do`
Send $(s, \mathsf{S}, \mathsf{GetTime})$ to LC and let $state = 3$.

Receive and process a request: Receive time, initialize the verifier: (B.51)
`if` $(s, \mathsf{S}, \mathsf{Time}, \underline{t_{\mathrm{s}}})$ is received from LC while $state = 3$, `do`
Send $(c, (\mathsf{C}, s, r), \mathsf{S}, \mathsf{Init})$ on $\mathsf{S}_{\mathrm{ver}}$ and let $state = 4$.

Receive and process a request: Verifier initialized, request verification: (B.52)
`if` $(c, (\mathsf{C}, s, r), \mathsf{S}, \mathsf{Inited})$ is received on $\mathsf{S}_{\mathrm{ver}}$ while $state = 4$, `do`
Send $(c, (\mathsf{C}, s, r), \mathsf{S}, \mathsf{Verify}, m_c, \sigma_c, pk_{\mathrm{c}}^{\mathsf{sig}})$ on $\mathsf{S}_{\mathrm{ver}}$ and let $state = 5$.

Receive and process a request: Decrypt session key: (B.53)
`if` $(c, (\mathsf{C}, s, r), \mathsf{S}, \mathsf{Verified}, \underline{b})$ is received on $\mathsf{S}_{\mathrm{ver}}$ while $state = 5$, `do`
Send $(s, \mathsf{Dec}, \varrho_k)$ to ENC and let $state = 6$.

Receive and process a request: Decrypt payload, execute protocol steps, relay request: (B.54)
`if` $(s, \mathsf{Plaintext}, (\mathsf{Key}, \underline{ptr}))$ is received from ENC while $state = 6$, `do`
Send $((s, c, r), \mathsf{Dec}, ptr, \varrho_{\mathrm{c}})$ to ENC.
Recv $((s, c, r), \mathsf{Plaintext}, \underline{p_{\mathrm{c}}})$ from ENC.
If $(\neg b) \vee (t_{\mathrm{c}} \leq t_{\min}) \vee (t_{\mathrm{c}} > t_{\mathrm{s}} + \mathsf{tol}^+) \vee (\exists t', c', sid'_{\mathrm{s}} \colon (t', r, c', sid'_{\mathrm{s}}) \in L)$, break.
While $|L| \geq \mathsf{cap}$:
  Let $t_{\min} = \min\{t' \mid (t', r', c', sid'_{\mathrm{s}}, ptr') \in L\}$ and $L = \{(t', r', c', sid'_{\mathrm{s}}, ptr') \in L \mid t' > t_{\min}\}$.
Generate an η-bit nonce sid_{s} randomly.
Insert $(t_{\mathrm{c}}, r, c, sid_{\mathrm{s}}, ptr)$ into L and L_{cor}.
Send $(s, c, sid_{\mathrm{s}}, \mathsf{Request}, p_{\mathrm{c}})$ to $E_{\mathrm{MX}}^{\mathrm{s}}$ and let $state = 1$.

Receive and process a response: Receive response payload, request key: (B.55)
`if` $(s, c, \underline{sid_{\mathrm{s}}}, \mathsf{Response}, \underline{p_{\mathrm{s}}})$ is received from $E_{\mathrm{MX}}^{\mathrm{s}}$ with $sid_{\mathrm{s}} \neq \varepsilon$ while $state > 0$, `do`
If $\neg\exists t, r, c\colon (t, r, c, sid_{\mathrm{s}}, ptr) \in L$,
  Send $(s, sid_{\mathrm{s}}, \mathsf{Response}_{\mathsf{Error}})$ to $E_{\mathrm{MX}}^{\mathrm{s}}$ and break.
Fetch $(\underline{t_{\mathrm{c}}}, \underline{r}, \underline{c}, sid_{\mathrm{s}}, \underline{ptr})$ from L.
Update the entry $(t_{\mathrm{c}}, r, c, sid_{\mathrm{s}}, ptr)$ in L to $(t_{\mathrm{c}}, r, c, \varepsilon, ptr)$.
Send $((s, c, r), \mathsf{Enc}, ptr, p_{\mathrm{s}})$ to ENC.
Recv $((s, c, r), \mathsf{Ciphertext}, \underline{\varrho_{\mathrm{s}}})$ from ENC.
Let $m_{\mathrm{s}} = (\mathsf{From}\colon c, \mathsf{To}\colon s, \mathsf{Ref}\colon r, \mathsf{Body}\colon \varrho_{\mathrm{s}})$.
Send $(s, (\mathsf{S}, c, r), \mathsf{GetKey})$ to $\mathsf{KS}^{\mathsf{sig}}$ and let $state = 7$.

Receive and process a response: Construct response message and request signature: (B.56)
`if` $(s, (\mathsf{S}, c, r), \mathsf{PublicKey}, \underline{pk_{\mathrm{s}}^{\mathsf{sig}}})$ is received from $\mathsf{KS}^{\mathsf{sig}}$ while $state = 7$, `do`
Send $(s, (\mathsf{S}, c, r), \mathsf{Sign}, m_s)$ on $\mathsf{S}_{\mathrm{sig}}$ and let $state = 8$.

Receive and process a response: Receive signature, send out message: (B.57)
`if` $(s, (\mathsf{S}, c, r), \text{Signature}, \underline{\sigma_s})$ is received on S_{sig} while $state = 8$, `do`
Send (m_s, σ_s) to A_S and let $state = 1$.

Reset the server: (B.58)
`if` (s, Reset) is received from A_S while $state > 0$, `do`
Let $t_{\min} = t_s + \text{tol}^+$, let $L = [\,]$ and $state = 1$.

Request to send response message for non-initialized server: (B.59)
`if` $(\underline{s}, \underline{c}, \underline{sid_s}, \text{Response}, \underline{p_s})$ is received from E^s_{MX} with $sid_s \neq \varepsilon$ while $state = 0$, `do`
Send $(s, sid_s, \text{Response}_{\text{Error}})$ to E^s_{MX} and halt.

Provide resources for corrupted signature scheme: (B.60)
`if` $(s, \underline{sid'_s}, \text{Corrupt}, 1^{\underline{n'}})$ is received from E^s_{MX} while $state > 0$, `do`
If $\neg\exists t', r', c', ptr' : (t', r', c', sid'_s, ptr') \in L_{cor}$, break.
Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, sid'_s, \underline{ptr'})$ from L_{cor}.
Send $(s, (\mathsf{S}, c', r'), \text{Resources}, 1^{n'})$ to KS^{sig}.

Corruption status: (B.61)
`if` $(s, sid_s, \text{Corrupted?})$ is received from E^s_{MX} while $state > 0$, `do`
If $\neg\exists t', r', c', ptr' : (t', r', c', sid_s, ptr') \in L_{cor}$, break.
Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, sid_s, \underline{ptr'})$ from L_{cor}.
Send $(s, (\mathsf{C}, c', r'), \text{Corrupted?})$ to KS^{ae}.
Recv $(s, (\mathsf{C}, c', r'), \text{Corrupted}, \underline{cor_1})$ from KS^{ae}.
Send $(s, (\mathsf{S}, c', r'), \text{Corrupted?})$ to KS^{sig}.
Recv $(s, (\mathsf{S}, c', r'), \text{Corrupted}, \underline{cor_2})$ from KS^{sig}.
Send $(s, (\mathsf{S}, c', r'), \mathsf{C}, \text{Corrupted?})$ to KS^{sig}.
Recv $(s, (\mathsf{S}, c', r'), \mathsf{C}, \text{Corrupted}, \underline{cor_3})$ from KS^{sig}.
Send $((s, c', r'), \text{Corrupted?}, ptr)$ to ENC.
Recv $((s, c', r'), \text{CorruptionState}, \underline{cor_4})$ from ENC.
Send $(s, sid_s, \text{Corrupted}, cor_1 \vee cor_2 \vee cor_3 \vee cor_4)$ to E^s_{MX}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.6. Server Functionality (PA) $\mathcal{P}^{\text{PA}}_{\mathsf{S}}$

Tapes: $\mathsf{S} \twoheadleftarrow\!\!\!\longrightarrow E^S_{\text{MX}}$, $\mathsf{S} \dashleftarrow\dashrightarrow A_\mathsf{S}$, $\mathsf{S} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{KS}^{\text{ae}}$, $\mathsf{S} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{KS}^{\text{sig}}$, $\mathsf{S} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{LC}$, $\mathsf{S} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{ENC}$, $\mathsf{S}_{\text{sig}} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{SIG}$, $\mathsf{S}_{\text{ver}} \longleftrightarrow\!\!\!\!\twoheadrightarrow \text{SIG}$

Variables and Initialization:

Variable	Type	Initial Value
$state, n, \text{cap}$	$\mathbb{N}$	0
U	$\{0,1\}^* \to \{0,1\}^*$	$\perp$
$c, s, r, H_r, m_c, m_s, \varrho_c, p_c, p_s, t_c, t_s, t_{\min}, \text{tol}^+$	$\{0,1\}^*$	$\perp$
L, L_{cor}	sets of 4-tuples of $\{0,1\}^*$	$[\,]$

Steps: `loop`

Initialization and Users: (B.62)
`if` $(\underline{s}, \text{Init}, \underline{U})$ is received from E_{SM} while $state = 0$, `do`
Send $(s, \text{GetParameters})$ to A_S.
Recv $(s, \text{Parameters}, \underline{\text{cap}}, \underline{\text{tol}^+})$ from A_S with $\text{cap} > 0$ and $\text{tol}^+ > 0$.
Send $(s, \mathsf{S}, \text{GetKey})$ to KS^{ae}.
Recv $(s, \mathsf{S}, \text{PublicKey}, \underline{pk^{\text{ae}}})$ from KS^{ae}.
Send $(s, \mathsf{S}, \text{GetTime})$ to LC.
Recv $(s, \mathsf{S}, \text{Time}, \underline{t_s})$ from LC.
Let $t_{\min} = t_s + \text{tol}^+$ and let $state = 1$.

Receive resources: (B.63)
`if` $(s, \text{Resources}, 1^{\underline{n}})$ is received from E_{SM} while $state > 0$, `do`
Break.

Receive and process a request: Request time: (B.64)

`if` $(\underline{m_{\mathrm{c}}}, \underline{\varrho_{\mathrm{c}}})$ is received from A_{S} with $m_{\mathrm{c}} = (\mathsf{From}\colon \underline{c}, \mathsf{To}\colon s, \mathsf{MsgID}\colon \underline{H_r}, \mathsf{Time}\colon \underline{t_{\mathrm{c}}}, \mathsf{Body}\colon \underline{p_{\mathrm{c}}})$ while $\mathit{state} > 0$, `do`

- If $|p_{\mathrm{c}}| + |\varrho_{\mathrm{c}}| \geq n - 1$, break.
- Let $n = 0$.
- Send $(s, \mathsf{S}, \mathsf{GetTime})$ to LC and let $\mathit{state} = 2$.

Receive and process a request: Receive time, request decryption: (B.65)

`if` $(s, \mathsf{S}, \mathsf{Time}, \underline{t_{\mathrm{s}}})$ is received from LC while $\mathit{state} = 2$, `do`

- Send $(s, \mathsf{Dec}, \varrho_{\mathrm{c}})$ to ENC and let $\mathit{state} = 3$.

Receive and process a request: Execute protocol steps, relay request: (B.66)

`if` $(s, \mathsf{Plaintext}, \underline{m'_{\mathrm{c}}})$ is received from ENC with $m'_{\mathrm{c}} = (\mathsf{SecMsgID}\colon \underline{r}, \mathsf{Pass}\colon \underline{pw}, \mathsf{MsgHash}\colon \underline{H_{m_{\mathrm{c}}}})$ while $\mathit{state} = 3$, `do`

- Send (GetRO, r) to RO.
- Recv $(\mathsf{RO}, \underline{H'_r})$ from RO.
- Send $(\mathsf{GetRO}, m_{\mathrm{c}})$ to RO.
- Recv $(\mathsf{RO}, \underline{H'_{m_{\mathrm{c}}}})$ from RO.
- If $(H_{m_{\mathrm{c}}} \neq H'_{m_{\mathrm{c}}}) \vee (H_r \neq H'_r) \vee (U(c) \neq pw) \vee (t_{\mathrm{c}} \leq t_{\mathrm{min}}) \vee (t_{\mathrm{c}} > t_{\mathrm{s}} + \mathrm{tol}^{+}) \vee (\exists t', c', \mathit{sid}'_{\mathrm{s}}\colon (t', r, c', \mathit{sid}'_{\mathrm{s}}) \in L)$, break.
- While $|L| \geq \mathrm{cap}$:
  - Let $t_{\mathrm{min}} = \min\{t' \mid (t', r', c', \mathit{sid}'_{\mathrm{s}}) \in L\}$ and $L = \{(t', r', c', \mathit{sid}'_{\mathrm{s}}) \in L \mid t' > t_{\mathrm{min}}\}$.
- Generate an η-bit nonce $\mathit{sid}_{\mathrm{s}}$ randomly.
- Insert $(t_{\mathrm{c}}, r, c, \mathit{sid}_{\mathrm{s}})$ into L and L_{cor}.
- Send $(s, \mathit{sid}_{\mathrm{s}}, \mathsf{Request}, p_{\mathrm{c}})$ to $E^{\mathrm{s}}_{\mathrm{MX}}$ and let $\mathit{state} = 1$.

Receive and process a response: Receive response payload, request key: (B.67)

`if` $(s, \underline{\mathit{sid}_{\mathrm{s}}}, \mathsf{Response}, \underline{p_{\mathrm{s}}})$ is received from $E^{\mathrm{s}}_{\mathrm{MX}}$ with $\mathit{sid}_{\mathrm{s}} \neq \varepsilon$ while $\mathit{state} > 0$, `do`

- If $\neg\exists t', r', c'\colon (t', r', c', \mathit{sid}_{\mathrm{s}}) \in L$,
  - Send $(s, \mathit{sid}_{\mathrm{s}}, \mathsf{Response}_{\mathsf{Error}})$ to $E^{\mathrm{s}}_{\mathrm{MX}}$ and break.
- Fetch $(\underline{t_{\mathrm{c}}}, \underline{r}, \underline{c}, \mathit{sid}_{\mathrm{s}})$ from L.
- Update the entry $(t_{\mathrm{c}}, r, c, \mathit{sid}_{\mathrm{s}})$ in L to $(t_{\mathrm{c}}, r, c, \varepsilon)$.
- Send (GetRO, r) to RO.
- Recv $(\mathsf{RO}, \underline{H_r})$ from RO.
- Send $(s, (\mathsf{S}, c, H_r), \mathsf{GetKey})$ to $\mathrm{KS}^{\mathrm{sig}}$ and let $\mathit{state} = 6$.

Receive and process a response: Construct response message and request signature: (B.68)

`if` $(s, (\mathsf{S}, c, H_r), \mathsf{PublicKey}, \underline{pk^{\mathrm{sig}}_{\mathrm{s}}})$ is received from KS while $\mathit{state} = 6$, `do`

- Let $m_{\mathrm{s}} = (\mathsf{From}\colon s, \mathsf{To}\colon c, \mathsf{Ref}\colon H_r, \mathsf{Body}\colon p_{\mathrm{s}})$.
- Send $(s, (\mathsf{S}, c, H_r), \mathsf{Sign}, m_s)$ on $\mathrm{S}_{\mathrm{sig}}$ and let $\mathit{state} = 7$.

Receive and process a response: Receive signature, send out message: (B.69)

`if` $(s, (\mathsf{S}, c, H_r), \mathsf{Signature}, \underline{\sigma_{\mathrm{s}}})$ is received on $\mathrm{S}_{\mathrm{sig}}$ while $\mathit{state} = 7$, `do`

- Send (m_s, σ_s) to A_{S} and let $\mathit{state} = 1$.

Reset the server: (B.70)

`if` (s, Reset) is received from A_{S} while $\mathit{state} > 0$, `do`

- Let $t_{\mathrm{min}} = t_{\mathrm{s}} + \mathrm{tol}^{+}$, let $L = [\,]$ and $\mathit{state} = 1$.

Request to send response message for non-initialized server: (B.71)

`if` $(\underline{s}, \underline{c}, \underline{\mathit{sid}_{\mathrm{s}}}, \mathsf{Response}, \underline{p_{\mathrm{s}}})$ is received from $E^{\mathrm{s}}_{\mathrm{MX}}$ with $\mathit{sid}_{\mathrm{s}} \neq \varepsilon$ while $\mathit{state} = 0$, `do`

- Send $(s, \mathit{sid}_{\mathrm{s}}, \mathsf{Response}_{\mathsf{Error}})$ to $E^{\mathrm{s}}_{\mathrm{MX}}$ and halt.

Provide resources for corrupted signature scheme: (B.72)

`if` $(s, \underline{\mathit{sid}'_{\mathrm{s}}}, \mathsf{Corrupt}, 1^{\underline{n'}})$ is received from $E^{\mathrm{s}}_{\mathrm{MX}}$ while $\mathit{state} > 0$, `do`

- If $\neg\exists t', r', c'\colon (t', r', c', \mathit{sid}_{\mathrm{s}}) \in L_{cor}$, break.
- Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, \mathit{sid}_{\mathrm{s}})$ from L_{cor}.
- Send $(s, (\mathsf{S}, c', H(r')), \mathsf{Resources}, 1^{n'})$ to $\mathrm{KS}^{\mathrm{sig}}$.

Corruption status: (B.73)
`if` $(s, \underline{sid_s}, \text{Corrupted?})$ is received from E^{s}_{MX} while $state > 0$, `do`
  If $\neg\exists t', r', c' : (t', r', c', sid_s) \in L_{cor}$, break.
  Fetch $(\underline{t'}, \underline{r'}, \underline{c'}, sid_s)$ from L_{cor}.
  Send $(s, (\mathsf{C}, c', H(r')), \text{Corrupted?})$ to KS^{ae}.
  Recv $(s, (\mathsf{C}, c', H(r')), \text{Corrupted}, \underline{cor_1})$ from KS^{ae}.
  Send $(s, (\mathsf{S}, c', H(r')), \text{Corrupted?})$ to KS^{sig}.
  Recv $(s, (\mathsf{S}, c', H(r')), \text{Corrupted}, \underline{cor_2})$ from KS^{sig}.
  Send $(s, (\mathsf{S}, c', H(r')), \mathsf{C}, \text{Corrupted?})$ to KS^{sig}.
  Recv $(s, (\mathsf{S}, c', H(r')), \mathsf{C}, \text{Corrupted}, \underline{cor_3})$ from KS^{sig}.
  Send $(s, sid_s, \text{Corrupted}, cor_1 \vee cor_2 \vee cor_3)$ to E^{s}_{MX}.

CheckAddress: Accept any message that is accepted by one of the steps.

B.2.7. Signature Key Store Functionality $\mathcal{F}_{\mathsf{KS}^{sig}}$

Tapes: $\mathsf{KS}^{sig} \twoheadleftarrow\!\!\longrightarrow \mathsf{SI}$, $\mathsf{KS}^{sig} \twoheadleftarrow\!\!\longrightarrow \mathsf{C}$, $\mathsf{KS}^{sig} \twoheadleftarrow\!\!\longrightarrow \mathsf{S}$, $\mathsf{KS}^{sig} \dashleftarrow\!\dashrightarrow A_{\mathsf{KS}^{sig}}$, $\mathsf{KS}^{sig}_{sig} \longleftarrow\!\!\twoheadrightarrow \mathsf{SIG}$, $\mathsf{KS}^{sig}_{ver} \longleftarrow\!\!\twoheadrightarrow \mathsf{SIG}$, $E_{sig} \longleftarrow\!\!\twoheadrightarrow \mathsf{SIG}$, $E_{ver} \longleftarrow\!\!\twoheadrightarrow \mathsf{SIG}$

Variables and Initialization:

Variable	Type	Initial Value
pk^{sig}	$\{0,1\}^*$	$\perp$
L	list of $\{0,1\}^*$	$[\,]$

Steps: `loop`

Request to get the key: (B.74)
`if` (GetKey) is received from $T \in \{\mathsf{C}, \mathsf{S}, \mathsf{SI}\}$, `do`
  Insert T into L.
  Send (GetKey, T) to $A_{\mathsf{KS}^{sig}}$.

Execute request to get the key: (B.75)
`if` $(\text{GetKey}, \underline{T})$ is received from $A_{\mathsf{KS}^{sig}}$, `do`
  If $T \notin L$, break.
  If $pk^{sig} = \perp$, send (Init) on KS^{sig}_{sig} and break.
  Delete T from L.
  Send $(\text{PublicKey}, pk^{sig})$ to T.

Store a generated key and notify the adversary: (B.76)
`if` $(\text{PublicKey}, \underline{pk^{sig}})$ is received on KS^{sig}_{sig}, `do`
  Send $(\text{PublicKey}, pk^{sig})$ to $A_{\mathsf{KS}^{sig}}$.

Is the signature functionality corrupted? (B.77)
`if` (Corrupted?) is received from $T \in \{\mathsf{C}, \mathsf{S}\}$, `do`
  If $pk^{sig} = \perp$, send $(\text{Corrupted}, \texttt{false})$ to T and break.
  Send (Corrupted?) on E_{sig}.
  Receive $(\underline{b})$ on E_{sig}.
  Send $(\text{Corrupted}, b)$ to T.

Is a verification functionality corrupted? (B.78)
`if` $(\underline{sid}, \text{Corrupted?})$ is received from $T \in \{\mathsf{C}, \mathsf{S}\}$, `do`
  If $pk^{sig} = \perp$, send $(sid, \text{Corrupted}, \texttt{false})$ to T and break.
  Send $(sid, \text{Corrupted?})$ on E_{ver}.
  Receive $(sid, \underline{b})$ on E_{ver}.
  Send $(sid, \text{Corrupted}, b)$ to T.

B.2.8. Public Key Encryption Key Store Functionality $\mathcal{F}_{\mathrm{KS}^{\mathrm{ae}}}$

Tapes: $\mathrm{KS}^{\mathrm{ae}} \twoheadleftarrow\!\!\longrightarrow \mathrm{C}$, $\mathrm{KS}^{\mathrm{ae}} \twoheadleftarrow\!\!\longrightarrow \mathrm{S}$, $\mathrm{KS}^{\mathrm{ae}} \leftarrow\!\!\dashrightarrow A_{\mathrm{KS}^{\mathrm{ae}}}$, $\mathrm{KS}^{\mathrm{ae}} \longleftrightarrow\!\!\!\!\twoheadrightarrow \mathrm{ENC}$

Variables and Initialization:

Variable	Type	Initial Value
pk^{ae}	$\{0,1\}^*$	$\bot$
L	list of $\{0,1\}^*$	[]

Steps: `loop`

Request to get the key: (B.79)
`if` $(\underline{p}, \mathrm{GetKey})$ is received from $T \in \{\mathrm{C}, \mathrm{S}\}$, `do`
- Insert (p, T) into L.
- Send (p, GetKey, T) to $A_{\mathrm{KS}^{\mathrm{ae}}}$.

Execute request to get the key: (B.80)
`if` $(\underline{p}, \mathrm{GetKey}, \underline{T})$ is received from $A_{\mathrm{KS}^{\mathrm{ae}}}$, `do`
- If $(p, T) \notin L$, break.
- If $pk^{\mathrm{ae}} = \bot$, send (KeyGen) to ENC and break.
- Delete (p, T) from L.
- Send $(p, \mathrm{PublicKey}, pk^{\mathrm{ae}})$ to T.

Store a generated key and notify the adversary: (B.81)
`if` $(\mathrm{PublicKey}, \underline{pk^{\mathrm{ae}}})$ is received from ENC, `do`
- Send $(\mathrm{PublicKey}, pk^{\mathrm{ae}})$ to $A_{\mathrm{KS}^{\mathrm{ae}}}$.

Is the public key encryption functionality corrupted? (B.82)
`if` $(\underline{p}, \mathrm{Corrupted?})$ is received from $T \in \{\mathrm{C}, \mathrm{S}\}$, `do`
- If $pk^{\mathrm{ae}} = \bot$, send $(p, \mathrm{Corrupted}, \mathtt{false})$ to T and break.
- Send (Corrupted?) to ENC.
- Receive $(\mathrm{CorruptionState}, \underline{x})$ on E_{sig}.
- Send $(p, \mathrm{Corrupted}, x)$ to T.

B.2.9. Signature Interface Functionality $\mathcal{P}_{\mathrm{SI}}(except)$

Parameters:

Description	Parameter	Type
Exception Function	*except*	$\{0,1\}^* \to \{\mathtt{true}, \mathtt{false}\}$

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

Variables and Initialization:

Variable	Type	Initial Value
$state, n$	$\mathbb{N}$	0
pk^{sig}	$\{0,1\}^*$	$\bot$

Steps: `loop`

Get resources from the environment to sign messages: (B.83)
`if` $(\mathrm{Resources}, 1^{\underline{n}})$ is received from E_{EI}, `do`
- Let $state = 1$.

Initialize the key and the verification functionality: (B.84)
`if` (Init) is received from A_{SI} while $state = 1$, `do`
- Send (GetKey) to $\mathrm{KS}^{\mathrm{sig}}$.
- Receive $(\mathrm{PublicKey}, \underline{pk^{\mathrm{sig}}})$ from $\mathrm{KS}^{\mathrm{sig}}$.
- Send (SI, Init) on $\mathrm{SI}_{\mathrm{ver}}$.
- Receive (SI, Inited) on $\mathrm{SI}_{\mathrm{ver}}$.
- Send $(\mathrm{PublicKey}, pk^{\mathrm{sig}})$ to A_{SI} and let $state = 2$.

Sign a message: (B.85)
`if` (Sign, $\underline{m}$) is received from A_{SI} while *state* $= 2$, `do`
 If $(except(m)) \vee (|m| > n)$, break.
 Let $n = 0$.
 Send (Sign, m) on SI_{sig}.
 Receive (Signature, $\underline{\sigma}$) on SI_{sig}.
 Send (Signature, σ) to A_{SI}.

Verify a message: (B.86)
`if` (Verify, $\underline{m}$, $\underline{\sigma}$) is received from A_{SI} while *state* $= 2$, `do`
 If $|m| > n$, break.
 Let $n = 0$.
 Send (SI, Verify, $m, \sigma, pk^{\text{sig}}$) on SI_{ver}.
 Receive (SI, Verified, $\underline{b}$) on SI_{ver}.
 Send (Verified, b) to A_{SI}.

B.2.10. Signature Interface Dummy Realization $\mathcal{P}_{\text{SI}}^{\text{dummy}}$

Tapes: $\text{SI} \twoheadleftarrow E_{\text{EI}}$

Steps: `loop`

Receive resources: (B.87)
`if` (Resources, $1^{\underline{n}}$) is received from E_{EI}, `do`
 Break.

B.2.11. Local Clock Functionality $\mathcal{F}_{\text{LC}}$

Tapes: $\text{LC} \twoheadleftarrow\!\!\rightarrow \text{C}$, $\text{LC} \twoheadleftarrow\!\!\rightarrow \text{S}$, $\text{LC} \dashleftarrow\!\dashrightarrow A_{\text{LC}}$

Variables and Initialization:

Variable	Type	Initial Value
t	$\mathbb{N}$	0

Steps: `loop`

Time Request: (B.88)
`if` (GetTime) is received from $T \in \{\text{C}, \text{S}\}$, `do`
 Send (GetTime) to A_{LC}.
 Recv (Time, $\underline{t'}$) from A_{LC}.
 If $t' \geq t$, let $t = t'$.
 Send (Time, t) to T.

B.2.12. Random Oracle Functionality $\mathcal{F}_{\text{RO}}$

Tapes: $\text{RO} \twoheadleftarrow\!\!\rightarrow \text{C}$, $\text{RO} \twoheadleftarrow\!\!\rightarrow \text{S}$, $\text{RO} \dashleftarrow\!\dashrightarrow A_{\text{RO}}$

Variables and Initialization:

Variable	Type	Initial Value
H	subset of $\{0,1\}^* \times \{0,1\}^\eta$	$\emptyset$
n	$\mathbb{N}$	0

Steps: `loop`

Retrieve a value: (B.89)
`if` (GetRO, $\underline{m}$) is received from $T \in \{\text{C}, \text{S}, A_{\text{RO}}\}$ with $(n > 0) \vee (T \neq A_{\text{RO}})$, `do`
 If $\neg\exists h\colon (m,h) \in H$,
  Generate an η-bit value h randomly.
  Let $H = H \cup \{(m,h)\}$.
 Fetch $(m, \underline{h})$ from H.
 If $T = A_{\text{RO}}$, let $n = n - 1$, else let $n = n + 1$.
 Send (RO, h) to T.

B.3. Simulators

B.3.1. Simulator (SA) $\mathcal{S}^{\text{SA}}_{\text{S2ME}}(\mathit{except})$

Parameters:

Description	Parameter	Type
Exception Function	*except*	$\{0,1\}^* \to \{$`true`, `false`$\}$

Tapes: C ⇠- - -⇢ A_{C}, S ⇠- - -⇢ A_{S}, A_{EI} ⇠- - -⇢ EI, A_{MX} ⇠- - -⇢ MX, A_{SM} ⇠- - -⇢ SM, plus the tapes between the adversary and the simulated machines (see below)

Variables and Initialization:

Variable	Type	Initial Value
sessions	subset of $(\{0,1\}^*)^4$	$\emptyset$
state, n, cap	associative array of $\mathbb{N}$	0
t, $t_{\min}$, tol^{+}	associative array of $\{0,1\}^*$	$\perp$
L	associative array of sets of 4-tuples of $\{0,1\}^*$	[ ]
cor	associative array of $\{$`true`, `false`$\}$	`false`

Steps: `loop`

Initialization of a server: (B.90)
`if` ($\underline{s}$, Init) is received from SM, `do`
 Run `processServerInit`(s) concurrently.

Request to send the request message: (B.91)
`if` ($\underline{s}, \underline{c}, \underline{sid_A}$, Request, $\underline{p_{\text{c}}}, \underline{l_{pw}}, \underline{n_{\text{c}}}$) is received from MX, `do`
 Run `processRequestRequest`$(s, c, sid_A, p_{\text{c}}, n_{\text{c}})$ concurrently.

Approval to send the request message: (B.92)
`if` $(\underline{m_{\text{c}}}, \underline{\sigma_{\text{c}}})$ is received from A_{S} with $m_{\text{c}} =$ (From: $\underline{c}$, To: $\underline{s}$, MsgID: $\underline{r}$, Time: $\underline{t_{\text{c}}}$, Body: $\underline{p_{\text{c}}}$) while $state[s] > 0$, `do`
 Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
 Run `processRequestApproval`(m_c, σ_c) concurrently.

Request to send the response message: (B.93)
`if` ($\underline{s}, \underline{c}, \underline{sid_A}$, Response, $\underline{p_{\text{s}}}$) is received from MX, `do`
 Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
 Run `processResponseRequest`$(s, c, sid_A, p_{\text{s}})$ concurrently.

Approval to send the response message: (B.94)
`if` $(\underline{m_{\text{s}}}, \underline{\sigma_{\text{s}}})$ is received from A_{C} with $m_{\text{s}} =$ (From: $\underline{s}$, To: $\underline{c}$, Ref: $\underline{r}$, Body: $\underline{p_{\text{s}}}$) while $state[c, s, r] = 2$, `do`
 Run `processResponseApproval`(m_{s}, σ_s) concurrently.

Reset the server: (B.95)
`if` ($\underline{s}$, Reset) is received from A_{S} while $state[s] > 0$, `do`
 Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
 Run `processServerReset`(s).

Resources for the Server: (B.96)
`if` $(\underline{s}, \text{Resources}, 1^{\underline{n'}})$ is received from SM while $state[s] > 0$, `do`
$n[s] = n'$.

Resources for Signing: (B.97)
`if` $(\underline{pid}, \underline{sid}, \text{Resources}, 1^{\underline{n'}})$ is received from EI, `do`
Send $(pid, sid, \text{Resources}, 1^{n'})$ to SI.

Resources for Corruption: (B.98)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \underline{x}, \text{Resources}, 1^{\underline{n'}})$ is received from MX with $x \in \{\mathsf{c}, \mathsf{s}\}$, `do`
Let $r = \texttt{nonce}(s, c, sid_A)$.
If $x = \mathsf{c}$, send $(c, (\mathsf{C}, s, r), \text{Resources}, 1^{n'})$ to KS^{sig}.
If $x = \mathsf{s}$, send $(s, (\mathsf{S}, c, r), \text{Resources}, 1^{n'})$ to KS^{sig}.

In addition, simulate

$$!\underline{\underline{\mathcal{F}_{\text{SIG}}}} \mid !\underline{\underline{\mathcal{P}_{\text{SI}(except)}}} \mid !\underline{\underline{\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid !\underline{\underline{\mathcal{F}_{\text{LC}}}}$$

and answer internal requests as well as request from the adversary to these machines, but in the following two cases, before the response messages are sent out, take additional actions:

Corruption of a signature scheme: (B.99)
`if` $(\underline{pid}, \underline{sid}, \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`
If $sid = (\mathsf{S}, \underline{c}, \underline{r})$,
Let $x = \mathsf{s}$ and let $s = pid$.
If $sid = (\mathsf{C}, \underline{s}, \underline{r})$,
Let $x = \mathsf{c}$ and let $c = pid$.
Let $sid_A = \texttt{sid}(s, c, r)$.
If $sid_A \neq \perp$,
Call $\texttt{corrupt}(s, c, sid_A, x)$.

Corruption of a verification scheme: (B.100)
`if` $(\underline{pid}, \underline{sid}, \underline{ssid}, \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`
If $(sid = (\mathsf{S}, \underline{c}, \underline{r})) \wedge (ssid = \mathsf{C})$,
Let $x = \mathsf{s}$ and let $s = pid$.
If $(sid = (\mathsf{C}, \underline{s}, \underline{r})) \wedge (ssid = \mathsf{S})$,
Let $x = \mathsf{c}$ and let $c = pid$.
Let $sid_A = \texttt{sid}(s, c, r)$.
If $sid_A \neq \perp$,
Call $\texttt{corrupt}(s, c, sid_A, x)$.

Functions:

Initialization of a server:
$\texttt{processServerInit}(s)$
Let $state[s] = 0$
Send $(s, \text{GetParameters})$ to A_{S}.
Recv $(s, \text{Parameters}, \underline{\text{cap}[s]}, \underline{\text{tol}^+[s]})$ from A_{S} with $\text{cap}[s] > 0$ or $\text{tol}^+[s] > 0$.
Let $t[s] = \texttt{getTime}(s, \mathsf{S})$.
Let $t_{\min}[s] = t[s] + \text{tol}^+[s]$, $n[s] = 0$, and $L[s] = [\,]$.
Let $state[s] = 1$.
Send $(s, \text{Init}_{\text{OK}})$ to SM.

Request to send the request message:
$\texttt{processRequestRequest}(s, c, sid_A, p_{\text{c}}, n_{\text{c}})$
Generate an η-bit nonce r randomly.
Let $state[c, s, r] = 1$ and let $n[c, s, r] = n_{\text{c}}$.
Let $sessions = sessions \cup \{(s, c, sid_A, r)\}$.
Let $t = \texttt{getTime}(c, (\mathsf{C}, s, r))$.
Let $m_{\text{c}} = (\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon r, \text{Time}\colon t, \text{Body}\colon p_{\text{c}})$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(c, (\mathsf{C}, s, r))$.
Let $\sigma_{\text{c}} = \texttt{sign}(c, (\mathsf{C}, s, r), m_{\text{c}})$.
Let $state[c, s, r] = 2$.
Send $(m_{\text{c}}, \sigma_{\text{c}})$ to A_{C}.

Approval to send the request message:
$\texttt{processRequestApproval}(m_c, \sigma_c)$
Let $(\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon r, \text{Time}\colon t_c, \text{Body}\colon p_c) = m_c$.
If $|p_c| > n[s]$, break.
Let $n[s] = 0$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(c, (\mathsf{C}, s, r))$.
Let $t[s] = \texttt{getTime}(s, \mathsf{S})$.
Let $b = \texttt{verify}(c, (\mathsf{C}, s, r), \mathsf{S}, m_c, \sigma_c, pk^{\text{sig}})$.
If $(\neg b) \vee (t_c \leq t_{\min}[s]) \vee (t_c > t[s] + \text{tol}^+[s]) \vee (\exists t', c', z'\colon (t', r, c', z') \in L[s])$, break.
While $|L[s]| \geq \text{cap}[s]$:
  Let $t_{\min}[s] = \min\{t' \mid (t', r', c', z') \in L[s]\}$.
  For $(\underline{t'}, \underline{r'}, \underline{c'}, \underline{z'}) \in L[s]$ with $(\neg z') \wedge (t' \leq t_{\min}[s])$,
    Send $(s, c', \texttt{sid}(s, c', r'), \text{Expire})$ to MX.
    Recv $(s, c', \texttt{sid}(s, c', r'), \text{Expire}_{\text{OK}})$ from MX.
  Let $L[s] = \{(t', r', c', z') \in L[s] \mid t' > t_{\min}[s]\}$.
Let $L[s] = L[s] \cup \{(t, r, c, \texttt{false})\}$.
Send $(s, c, \texttt{sid}(s, c, r), \text{Request}_{\text{OK}}, p_c, \varepsilon)$ to MX.

Request to send the response message:
$\texttt{processResponseRequest}(s, c, sid_A, p_s)$
Let $r = \text{nonce}(s, c, sid_A)$.
Fetch $(\underline{t}, r, c, \texttt{false})$ from $L[s]$.
Update $(t, r, c, \texttt{false})$ in $L[s]$ to $(t, r, c, \texttt{true})$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(s, (\mathsf{S}, c, r))$.
Let $m_s = (\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon r, \text{Body}\colon p_s)$.
Let $\sigma_s = \texttt{sign}(s, (\mathsf{S}, c, r), m_s)$.
Send (m_s, σ_s) to A_{S}.

Approval to send the response message:
$\texttt{processResponseApproval}(m_s, \sigma_s)$
Let $(\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon r, \text{Body}\colon p_s) = m_s$.
Let $state[c, s, r] = 3$.
If $|p_s| > n[c, s, r]$, break.
Let $pk^{\text{sig}} = \texttt{getSigKey}(s, (\mathsf{S}, c, r))$.
Let $b = \texttt{verify}(s, (\mathsf{S}, c, r), \mathsf{C}, m_s, \sigma_s, pk^{\text{sig}})$.
If $\neg b$, break.
Send $(s, c, \texttt{sid}(s, c, r), \text{Response}_{\text{OK}}, p_s)$ to MX.

Reset of the server:
$\texttt{processServerReset}(s)$
For $(\underline{t}, \underline{r}, \underline{c}, \underline{z}) \in L[s]$ with $\neg z$,
  Send $(s, c, \texttt{sid}(s, c, r), \text{Expire})$ to MX.
  Recv $(s, c, \texttt{sid}(s, c, r), \text{Expire}_{\text{OK}})$ from MX.
Let $t_{\min}[s] = t[s] + \text{tol}^+[s]$ and $L[s] = [\,]$.

Get the time of a principal:
$\texttt{getTime}(pid, sid)$
Send $(pid, sid, \text{GetTime})$ to LC.
Recv $(pid, sid, \text{Time}, \underline{t})$ from LC.
Return t.

Get a key from the signature key store:
$\texttt{getSigKey}(pid, sid)$
Send $(pid, sid, \text{GetKey})$ to KS^{sig}.
Recv $(pid, sid, \text{PublicKey}, \underline{pk^{\text{sig}}})$ from KS^{sig}.
Return pk^{sig}.

Get a signature:
$\texttt{sign}(pid, sid, m)$
Send $(pid, sid, \text{Sign}, m)$ to SIG.
Recv $(pid, sid, \text{Signature}, \underline{\sigma})$ from SIG.
Return σ.

Verify a signature:
`verify`$(pid, sid, ssid, m, \sigma, pk^{\text{sig}})$
  Send $(pid, sid, ssid, \text{Init})$ to SIG.
  Recv $(pid, sid, ssid, \text{Inited})$ from SIG.
  Send $(pid, sid, ssid, \text{Verify}, m, \sigma, pk^{\text{sig}})$ to SIG.
  Recv $(pid, sid, ssid, \text{Verified}, \underline{b})$ from SIG.
  Return b.

Corrupt a session:
`corrupt`(s, c, sid_A, x)
  If $\neg cor[s, c, sid_A, x]$,
    Send $(s, c, sid_A, \text{Corrupt}, x)$ to MX.
    Recv $(s, c, sid_A, \text{Corrupt}_{\text{OK}}, x)$ from MX.
  Let $cor[s, c, sid_A, x] = \texttt{true}$.

Retrieve the nonce of a session:
`nonce`(s, c, sid_A)
  Fetch $(s, c, sid_A, \underline{r})$ from *sessions*.
  Return r.

Retrieve the session id of a session:
`sid`(s, c, r)
  Fetch $(s, c, \underline{sid_A}, r)$ from *sessions*.
  Return sid_A.

B.3.2. Simulator (CSA) $\mathcal{S}^{\text{CSA}}_{\text{S2ME}}(except)$

Parameters:

Description	Parameter	Type
Exception Function	*except*	$\{0,1\}^* \to \{\texttt{true}, \texttt{false}\}$

Tapes: C $\leftarrow\!\!\leftarrow\!\!-\!\!-\!\!\rightarrow$ A_{C}, S $\leftarrow\!\!\leftarrow\!\!-\!\!-\!\!\rightarrow$ A_{S}, A_{EI} $\leftarrow\!\!-\!\!-\!\!\rightarrow$ EI, A_{MX} $\leftarrow\!\!-\!\!-\!\!\rightarrow$ MX, A_{SM} $\leftarrow\!\!-\!\!-\!\!\rightarrow$ SM,
plus the tapes between the adversary and the simulated machines (see below)

Variables and Initialization:

Variable	Type	Initial Value
sessions	subset of $(\{0,1\}^*)^4$	$\emptyset$
$state, n, \text{cap}$	associative array of $\mathbb{N}$	0
$t, t_{\min}, \text{tol}^+, ptrs$	associative array of $\{0,1\}^*$	$\perp$
L	associative array of sets of 5-tuples of $\{0,1\}^*$	[ ]
cor	associative array of $\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Initialization of a server: (B.101)
`if` $(\underline{s}, \text{Init})$ is received from SM, `do`
  Run `processServerInit`(s) concurrently.

Request to send the request message: (B.102)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \text{Request}, \underline{p_{\text{c}}}, \underline{l_{pw}}, \underline{n_{\text{c}}})$ is received from MX, `do`
  Run `processRequestRequest`$(s, c, sid_A, p_{\text{c}}, n_{\text{c}})$ concurrently.

Approval to send the request message: (B.103)
`if` $(\underline{m_{\text{c}}}, \underline{\sigma_{\text{c}}})$ is received from A_{S} with $m_{\text{c}} = (\text{From}: \underline{c}, \text{To}: \underline{s}, \text{MsgID}: \underline{r}, \text{Time}: \underline{t_{\text{c}}}, \text{Key}: \underline{\varrho_k}, \text{Body}: \underline{\varrho_{\text{c}}})$ while $state[s] > 0$, `do`
  Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
  Run `processRequestApproval`(m_c, σ_c) concurrently.

Request to send the response message: (B.104)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \text{Response}, \underline{\varrho_{\text{s}}})$ is received from MX, `do`
  Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
  Run `processResponseRequest`$(s, c, sid_A, \varrho_{\text{s}})$ concurrently.

Approval to send the response message: (B.105)
`if` $(\underline{m_s}, \underline{\sigma_s})$ is received from A_C with $m_s = (\text{From}: \underline{s}, \text{To}: \underline{c}, \text{Ref}: \underline{r}, \text{Body}: \underline{\varrho_s})$ while $state[c, s, r] = 2$, `do`
Run `processResponseApproval`(m_s, σ_s) concurrently.

Reset the server: (B.106)
`if` $(\underline{s}, \text{Reset})$ is received from A_S while $state[s] > 0$, `do`
Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
Run `processServerReset`(s).

Resources for the Server: (B.107)
`if` $(\underline{s}, \text{Resources}, 1^{\underline{n'}})$ is received from SM while $state[s] > 0$, `do`
$n[s] = n'$.

Resources for Signing: (B.108)
`if` $(\underline{pid}, \underline{sid}, \text{Resources}, 1^{\underline{n'}})$ is received from EI, `do`
Send $(pid, sid, \text{Resources}, 1^{n'})$ to SI.

Resources for Corruption: (B.109)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \underline{x}, \text{Resources}, 1^{\underline{n'}})$ is received from MX with $x \in \{\text{c}, \text{s}\}$, `do`
Let $r = \texttt{nonce}(s, c, sid_A)$.
If $x = \text{c}$, send $(c, (\text{C}, s, r), \text{Resources}, 1^{n'})$ to KS^{sig}.
If $x = \text{s}$, send $(s, (\text{S}, c, r), \text{Resources}, 1^{n'})$ to KS^{sig}.

In addition, simulate

$$!\underline{\underline{\mathcal{F}_{\text{SIG}}}} \mid \mathcal{F}_{\text{ENC}} \mid !\underline{\underline{\mathcal{P}_{\text{SI}(except)}}} \mid !\underline{\underline{\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid !\underline{\underline{\mathcal{F}_{\text{KS}^{\text{ae}}}}} \mid !\underline{\underline{\mathcal{F}_{\text{LC}}}}$$

and answer internal requests as well as request from the adversary to these machines, but but in the following two cases, before the response messages are sent out, take additional actions:

Corruption of a public key encryption scheme: (B.110)
`if` $(\underline{pid}, \text{Algorithms}, \underline{enc}, \underline{dec}, \underline{pk}, \texttt{true})$ is received from A to ENC and if ENC responds with (pid, Ack), `do`
Let $cor[pid] = \texttt{true}$.
For $(pid, \underline{c}, \underline{sid_A}, \underline{r})$ in *sessions*,
Call `corrupt`(pid, c, sid_A).

Corruption of a signature scheme: (B.111)
`if` $(\underline{pid}, \underline{sid}, \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`
If $sid = (\text{S}, \underline{c}, \underline{r})$,
Let $x = \text{s}$ and let $s = pid$.
If $sid = (\text{C}, \underline{s}, \underline{r})$,
Let $x = \text{c}$ and let $c = pid$.
Let $sid_A = \texttt{sid}(s, c, r)$.
If $sid_A \neq \perp$,
Call `corrupt`(s, c, sid_A, x).

Corruption of a verification scheme: (B.112)
`if` $(\underline{pid}, \underline{sid}, \underline{ssid}, \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`
If $(sid = (\text{S}, \underline{c}, \underline{r})) \wedge (ssid = \text{C})$,
Let $x = \text{s}$ and let $s = pid$.
If $(sid = (\text{C}, \underline{s}, \underline{r})) \wedge (ssid = \text{S})$,
Let $x = \text{c}$ and let $c = pid$.
Let $sid_A = \texttt{sid}(s, c, r)$.
If $sid_A \neq \perp$,
Call `corrupt`(s, c, sid_A, x).

Functions:

Initialization of a server:
`processServerInit`(s)
Let $state[s] = 0$
Send $(s, \text{GetParameters})$ to A_S.
Recv $(s, \text{Parameters}, \underline{\text{cap}[s]}, \underline{\text{tol}^+[s]})$ from A_S with $\text{cap}[s] > 0$ or $\text{tol}^+[s] > 0$.
Call $\texttt{getEncKey}_{\texttt{ae}}(s, \text{S})$.
Let $t[s] = \texttt{getTime}(s, \text{S})$.
Let $t_{\min}[s] = t[s] + \text{tol}^+[s]$, $n[s] = 0$, and $L[s] = [\,]$.
Let $state[s] = 1$.
Send $(s, \text{Init}_{\text{OK}})$ to SM.

Request to send the request message:
`processRequestRequest`(s, c, sid_A, p_c, n_c)
If $cor[s]$,
Call $\texttt{corrupt}(s, c, sid_A)$.
Generate an η-bit nonce r randomly.
Let $state[c, s, r] = 1$ and let $n[c, s, r] = n_c$.
Let $sessions = sessions \cup \{(s, c, sid_A, r)\}$.
Let $t = \texttt{getTime}(c, (\text{C}, s, r))$.
Let $ptr = \texttt{getEncKey}_{\texttt{se}}(s, c, r)$.
Let $ptrs[s, c, r] = ptr$.
If $\texttt{isCorrupt}(s, c, r, ptr)$,
Call $\texttt{corrupt}(s, c, sid_A, \text{c})$.
If $cor[s, c, sid_A, \text{c}]$,
Send $(s, c, sid_A, \text{Reveal}, \text{c})$ to MX.
Recv $(s, c, sid_A, \text{Reveal}, \text{c}, \underline{p_c}, \underline{pw})$ from MX.
Let $pk^{\text{ae}} = \texttt{getEncKey}_{\texttt{ae}}(s, (\text{C}, c, r))$.
Let $\varrho_k = \texttt{encrypt}_{\texttt{ae}}(s, (\text{C}, c, r), pk^{\text{ae}}, (\text{Key}, ptr))$.
Let $\varrho_c = \texttt{encrypt}_{\texttt{se}}(s, c, r, ptr, p_c)$.
Let $m_c = (\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon r, \text{Time}\colon t, \text{Key}\colon \varrho_k, \text{Body}\colon \varrho_c)$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(c, (\text{C}, s, r))$.
Let $\sigma_c = \texttt{sign}(c, (\text{C}, s, r), m_c)$.
Let $state[c, s, r] = 2$.
Send (m_c, σ_c) to A_C.

Approval to send the request message:
`processRequestApproval`(ϱ_c, σ_c)
Let $(\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon r, \text{Time}\colon t_c, \text{Key}\colon \varrho_k, \text{Body}\colon \varrho_c) = m_c$.
If $|p_c| > n[s]$, break.
Let $n[s] = 0$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(c, (\text{C}, s, r))$.
Let $t[s] = \texttt{getTime}(s, \text{S})$.
Let $b = \texttt{verify}(c, (\text{C}, s, r), \text{S}, m_c, \sigma_c, pk^{\text{sig}})$.
Let $(\text{Key}, ptr) = \texttt{decrypt}_{\texttt{ae}}(s, \varrho_k)$.
Let $p_c = \texttt{decrypt}_{\texttt{se}}(s, c, r, ptr, \varrho_c)$.
If $(\neg b) \lor (t_c \leq t_{\min}[s]) \lor (t_c > t[s] + \text{tol}^+[s]) \lor (\exists t', c', ptr', z'\colon (t', r, c', ptr', z') \in L[s])$, break.
While $|L[s]| \geq \text{cap}[s]$:
Let $t_{\min}[s] = \min\{t' \mid (t', r', c', ptr', z') \in L[s]\}$.
For $(\underline{t'}, \underline{r'}, \underline{c'}, \underline{ptr'}, \underline{z'}) \in L[s]$ with $(\neg z') \land (t' \leq t_{\min}[s])$,
Send $(s, c', \texttt{sid}(s, c', r'), \text{Expire})$ to MX.
Recv $(s, c', \texttt{sid}(s, c', r'), \text{Expire}_{\text{OK}})$ from MX.
Let $L[s] = \{(t', r', c', ptr', z') \in L[s] \mid t' > t_{\min}[s]\}$.
Let $L[s] = L[s] \cup \{(t, r, c, ptr, \texttt{false})\}$.
Send $(s, c, \texttt{sid}(s, c, r), \text{Request}_{\text{OK}}, p_c, \varepsilon)$ to MX.

Request to send the response message:
$\texttt{processResponseRequest}(s, c, sid_A, p_s)$

Let $r = \texttt{nonce}(s, c, sid_A)$.
Fetch $(\underline{t}, r, c, ptr, \texttt{false})$ from $L[s]$.
Update $(t, r, c, ptr, \texttt{false})$ in $L[s]$ to $(t, r, c, ptr, \texttt{true})$.
If $\texttt{isCorrupt}(s, c, r, ptr)$,
 Call $\texttt{corrupt}(s, c, sid_A, \mathrm{s})$.
If $cor[s, c, sid_A, \mathrm{s}]$,
 Send $(s, c, sid_A, \mathrm{Reveal}, \mathrm{s})$ to MX.
 Recv $(s, c, sid_A, \mathrm{Reveal}, \mathrm{s}, \underline{p_s}, \underline{pw})$ from MX.
Let $\varrho_s = \texttt{encrypt}_{\texttt{se}}(s, c, r, ptr, p_s)$.
Let $pk^{\mathrm{sig}} = \texttt{getSigKey}(s, (\mathrm{S}, c, r))$.
Let $m_s = (\mathrm{From}\colon c, \mathrm{To}\colon s, \mathrm{Ref}\colon r, \mathrm{Body}\colon \varrho_s)$.
Let $\sigma_s = \texttt{sign}(s, (\mathrm{S}, c, r), m_s)$.
Send (m_s, σ_s) to A_S.

Approval to send the response message:
$\texttt{processResponseApproval}(m_s, \sigma_s)$

Let $(\mathrm{From}\colon c, \mathrm{To}\colon s, \mathrm{Ref}\colon r, \mathrm{Body}\colon \varrho_s) = m_s$.
Let $state[c, s, r] = 3$.
If $|\varrho_s| > n[c, s, r]$, break.
Let $pk^{\mathrm{sig}} = \texttt{getSigKey}(s, (\mathrm{S}, c, r))$.
Let $b = \texttt{verify}(s, (\mathrm{S}, c, r), \mathrm{C}, m_s, \sigma_s, pk^{\mathrm{sig}})$.
If $\neg b$, break.
Let $p_s = \texttt{decrypt}_{\texttt{se}}(s, c, r, ptrs[s, c, r], \varrho_s)$.
Send $(s, c, \texttt{sid}(s, c, r), \mathrm{Response}_{\mathrm{OK}}, p_s)$ to MX.

Reset of the server:
$\texttt{processServerReset}(s)$

For $(\underline{t}, \underline{r}, \underline{c}, \underline{ptr}, \underline{z}) \in L[s]$ with $\neg z$,
 Send $(s, c, \texttt{sid}(s, c, r), \mathrm{Expire})$ to MX.
 Recv $(s, c, \texttt{sid}(s, c, r), \mathrm{Expire}_{\mathrm{OK}})$ from MX.
Let $t_{\min}[s] = t[s] + \mathrm{tol}^{+}[s]$ and $L[s] = [\,]$.

Get the time of a principal:
$\texttt{getTime}(pid, sid)$

Send $(pid, sid, \mathrm{GetTime})$ to LC.
Recv $(pid, sid, \mathrm{Time}, \underline{t})$ from LC.
Return t.

Get a key from the signature key store:
$\texttt{getSigKey}(pid, sid)$

Send $(pid, sid, \mathrm{GetKey})$ to $\mathrm{KS}^{\mathrm{sig}}$.
Recv $(pid, sid, \mathrm{PublicKey}, \underline{pk^{\mathrm{sig}}})$ from $\mathrm{KS}^{\mathrm{sig}}$.
Return pk^{sig}.

Get a key from the public key encryption key store:
$\texttt{getEncKey}_{\texttt{ae}}(pid, sid)$

Send $(pid, sid, \mathrm{GetKey})$ to $\mathrm{KS}^{\mathrm{ae}}$.
Recv $(pid, sid, \mathrm{PublicKey}, \underline{pk^{\mathrm{ae}}})$ from $\mathrm{KS}^{\mathrm{ae}}$.
Return pk^{ae}.

Get a key from the public key encryption key store:
$\texttt{getEncKey}_{\texttt{se}}(s, c, r)$

Send $((s, c, r), \mathrm{KeyGen})$ to ENC.
Recv $((s, c, r), \mathrm{KeyGen}, \underline{ptr})$ from ENC.
Return ptr.

Get a signature:
$\texttt{sign}(pid, sid, m)$

Send $(pid, sid, \mathrm{Sign}, m)$ to SIG.
Recv $(pid, sid, \mathrm{Signature}, \underline{\sigma})$ from SIG.
Return σ.

Verify a signature:
$\texttt{verify}(pid, sid, ssid, m, \sigma, pk^{\text{sig}})$
    Send $(pid, sid, ssid, \text{Init})$ to SIG.
    Recv $(pid, sid, ssid, \text{Inited})$ from SIG.
    Send $(pid, sid, ssid, \text{Verify}, m, \sigma, pk^{\text{sig}})$ to SIG.
    Recv $(pid, sid, ssid, \text{Verified}, \underline{b})$ from SIG.
    Return b.

Encrypt a plaintext (public key encryption):
$\texttt{encrypt}_{\texttt{ae}}(pid, sid, pk^{\text{ae}}, m)$
    Send (pid, sid, Init) to ENC.
    Recv $(pid, sid, \text{Inited})$ from ENC.
    Send $(pid, sid, \text{Enc}, pk^{\text{ae}}, m)$ to ENC.
    Recv $(pid, sid, \text{Ciphertext}, \underline{\varrho})$ from ENC.
    Return ϱ.

Decrypt a ciphertext (public key encryption):
$\texttt{decrypt}_{\texttt{ae}}(pid, \varrho)$
    Send $(pid, \text{Dec}, \varrho)$ to ENC.
    Recv $(pid, \text{Plaintext}, m)$ from ENC.
    Return m.

Encrypt a plaintext (symmetric encryption):
$\texttt{encrypt}_{\texttt{se}}(s, c, r, ptr, m)$
    Send $((s, c, r), \text{Enc}, ptr, m)$ to ENC.
    Recv $((s, c, r), \text{Ciphertext}, \underline{\varrho})$ from ENC.
    Return ϱ.

Decrypt a ciphertext (symmetric encryption):
$\texttt{decrypt}_{\texttt{se}}(pid, ptr, \varrho)$
    Send $(pid, \text{Dec}, ptr, \varrho)$ to ENC.
    Recv $(pid, \text{Plaintext}, \underline{m})$ from ENC.
    Return m.

Corruption status of a symmetric key:
$\texttt{isCorrupt}(s, c, r, ptr)$
    Send $((s, c, r), \text{Corrupted?}, ptr)$ to ENC.
    Recv $((s, c, r), \text{CorruptionState}, \underline{b})$ from ENC.
    Return b.

Corrupt a session:
$\texttt{corrupt}(s, c, sid_A)$
    Call $\texttt{corrupt}(s, c, sid_A, \mathrm{c})$.
    Call $\texttt{corrupt}(s, c, sid_A, \mathrm{s})$.

Corrupt a session:
$\texttt{corrupt}(s, c, sid_A, x)$
    If $\neg cor[s, c, sid_A, x]$,
        Send $(s, c, sid_A, \text{Corrupt}, x)$ to MX.
        Recv $(s, c, sid_A, \text{Corrupt}_{\text{OK}}, x)$ from MX.
    Let $cor[s, c, sid_A, x] = \texttt{true}$.

Retrieve the nonce of a session:
$\texttt{nonce}(s, c, sid_A)$
    Fetch $(s, c, sid_A, \underline{r})$ from *sessions*.
    Return r.

Retrieve the session id of a session:
$\texttt{sid}(s, c, r)$
    Fetch $(s, c, \underline{sid_A}, r)$ from *sessions*.
    Return sid_A.

B.3.3. Simulator (PA) $\mathcal{S}^{\mathrm{PA}}_{\mathrm{S2ME}}(\mathit{except})$

Parameters:

Description	Parameter	Type
Exception Function	*except*	$\{0,1\}^* \to \{\mathtt{true}, \mathtt{false}\}$

Tapes: C $\leftrightarrow$ A_{C}, S $\leftrightarrow$ A_{S}, A_{EI} $\leftrightarrow$ EI, A_{MX} $\leftrightarrow$ MX, A_{SM} $\leftrightarrow$ SM, plus the tapes between the adversary and the simulated machines (see below)

Variables and Initialization:

Variable	Type	Initial Value
sessions	subset of $(\{0,1\}^*)^5$	$\emptyset$
$\mathit{state}, n, \mathrm{cap}$	associative array of $\mathbb{N}$	0
$t, t_{\min}, \mathrm{tol}^{+}$	associative array of $\{0,1\}^*$	$\perp$
L	associative array of sets of 5-tuples of $\{0,1\}^*$	[ ]
cor	associative array of $\{\mathtt{true}, \mathtt{false}\}$	`false`

Steps: `loop`

Initialization of a server: (B.113)
`if` $(\underline{s}, \mathrm{Init})$ is received from SM, `do`
Run $\mathtt{processServerInit}(s)$ concurrently.

Request to send the request message: (B.114)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \mathrm{Request}, \underline{p_{\mathrm{c}}}, \underline{l_{pw}}, \underline{n_{\mathrm{c}}})$ is received from MX, `do`
Run $\mathtt{processRequestRequest}(s, c, sid_A, p_{\mathrm{c}}, l_{pw}, n_{\mathrm{c}})$ concurrently.

Approval to send the request message: (B.115)
`if` $(\underline{m_{\mathrm{c}}}, \underline{\sigma_{\mathrm{c}}})$ is received from A_{S} with $m_{\mathrm{c}} = (\mathrm{From}\colon \underline{c}, \mathrm{To}\colon \underline{s}, \mathrm{MsgID}\colon \underline{r}, \mathrm{Time}\colon \underline{t_{\mathrm{c}}}, \mathrm{Body}\colon \underline{p_{\mathrm{c}}})$ while $\mathit{state}[s] > 0$, `do`
Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
Run $\mathtt{processRequestApproval}(m_c, \sigma_c)$ concurrently.

Request to send the response message: (B.116)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \mathrm{Response}, \underline{p_{\mathrm{s}}})$ is received from MX, `do`
Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
Run $\mathtt{processResponseRequest}(s, c, sid_A, p_{\mathrm{s}})$ concurrently.

Approval to send the response message: (B.117)
`if` $(\underline{m_{\mathrm{s}}}, \underline{\sigma_{\mathrm{s}}})$ is received from A_{C} with $m_{\mathrm{s}} = (\mathrm{From}\colon \underline{s}, \mathrm{To}\colon \underline{c}, \mathrm{Ref}\colon \underline{H_r}, \mathrm{Body}\colon \underline{p_{\mathrm{s}}})$ while $\mathit{state}[c, s, H_r] = 2$, `do`
Run $\mathtt{processResponseApproval}(m_{\mathrm{s}}, \sigma_s)$ concurrently.

Reset the server: (B.118)
`if` $(\underline{s}, \mathrm{Reset})$ is received from A_{S} while $\mathit{state}[s] > 0$, `do`
Cancel any concurrent runs of `processRequestApproval` or `processResponseRequest` with server identity s.
Run $\mathtt{processServerReset}(s)$.

Resources for the Server: (B.119)
`if` $(\underline{s}, \mathrm{Resources}, 1^{\underline{n'}})$ is received from SM while $\mathit{state}[s] > 0$, `do`
$n[s] = n'$.

Resources for Signing: (B.120)
`if` $(\underline{pid}, \underline{sid}, \mathrm{Resources}, 1^{\underline{n'}})$ is received from EI, `do`
Send $(pid, sid, \mathrm{Resources}, 1^{n'})$ to SI.

Resources for Corruption: (B.121)
`if` $(\underline{s}, \underline{c}, \underline{sid_A}, \mathrm{s}, \mathrm{Resources}, 1^{\underline{n'}})$ is received from MX, `do`
Let $r = \mathtt{nonce}(s, c, sid_A)$.
Send $(s, (\mathrm{S}, c, \mathtt{hash}(r)), \mathrm{Resources}, 1^{n'})$ to $\mathrm{KS}^{\mathrm{sig}}$.

In addition, simulate

$$!\underline{\underline{\mathcal{F}_{\text{SIG}}}} \mid \mathcal{F}_{\text{ENC}} \mid !\underline{\underline{\mathcal{P}_{\text{SI}(except)}}} \mid !\underline{\underline{\mathcal{F}_{\text{KS}^{\text{sig}}}}} \mid !\underline{\underline{\mathcal{F}_{\text{KS}^{\text{ae}}}}} \mid !\underline{\underline{\mathcal{F}_{\text{LC}}}} \mid \mathcal{F}_{\text{RO}}$$

and answer internal requests as well as request from the adversary to these machines, but in the following two cases, take additional actions:

Corruption of a public key encryption scheme: (B.122)

`if` $(\underline{pid}, \text{Algorithms}, \underline{enc}, \underline{dec}, \underline{pk}, \texttt{true})$ is received from A to ENC and if ENC responds with (pid, Ack), `do`

Let $cor[pid] = \texttt{true}$.
For $(pid, \underline{c}, \underline{sid_A}, \underline{r}, \underline{so})$ in *sessions*,
Call $\texttt{corrupt}(pid, c, sid_A)$.

Corruption of a signature scheme: (B.123)

`if` $(\underline{s}, (\text{S}, \underline{c}, \underline{H_r}), \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`

Let $cor[s, c, H_r] = \texttt{true}$.
For sid_A in $\texttt{sids}(s, c, H_r)$,
Call $\texttt{corrupt}(s, c, sid_A, \text{s})$.

Corruption of a verification scheme: (B.124)

`if` $(\underline{s}, (\text{S}, \underline{c}, \underline{H_r}), \text{C}, \text{Corrupted}, \underline{p})$ is sent from SIG to A, `do`

Let $cor[s, c, H_r] = \texttt{true}$.
For sid_A in $\texttt{sids}(s, c, H_r)$,
Call $\texttt{corrupt}(s, c, sid_A, \text{s})$.

Functions:

Initialization of a server:

$\texttt{processServerInit}(s)$

Let $state[s] = 0$
Send $(s, \text{GetParameters})$ to A_S.
Recv $(s, \text{Parameters}, \underline{\text{cap}[s]}, \underline{\text{tol}^+[s]})$ from A_S with $\text{cap}[s] > 0$ or $\text{tol}^+[s] > 0$.
Call $\texttt{getEncKey}_{\texttt{ae}}(s, \text{S})$.
Let $t[s] = \texttt{getTime}(s, \text{S})$.
Let $t_{\min}[s] = t[s] + \text{tol}^+[s]$, $n[s] = 0$, and $L[s] = [\,]$.
Let $state[s] = 1$.
Send $(s, \text{Init}_{\text{OK}})$ to SM.

Request to send the request message:

$\texttt{processRequestRequest}(s, c, sid_A, p_\text{c}, l_{pw}, n_\text{c})$

If $cor[s]$,
Call $\texttt{corrupt}(s, c, sid_A)$.
Generate an η-bit nonce r randomly.
Let $H_r = \texttt{hash}(r)$.
Let $state[c, s, H_r] = 1$ and let $n[c, s, H_r] = n_\text{c}$.
Let $sessions = sessions \cup \{(s, c, sid_A, r, \texttt{false})\}$.
Let $t = \texttt{getTime}(c, (\text{C}, s, H_r))$.
Let $pk^{\text{ae}} = \texttt{getEncKey}_{\texttt{ae}}(s, (\text{C}, c, H_r))$.
Let $m_\text{c} = (\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon H_r, \text{Time}\colon t, \text{Body}\colon p_\text{c})$.
Let $H_{m_\text{c}} = \texttt{hash}(m_\text{c})$.
If $cor[s]$,
Send $(s, c, sid_A, \text{Reveal}, \text{c})$ to MX.
Recv $(s, c, sid_A, \text{Reveal}, \text{c}, p_\text{c}, \underline{pw})$ from MX,
else,
Generate an l_{pw}-bit bitstring pw randomly.
Let $m'_\text{c} = (\text{SecMsgID}\colon r, \text{Pass}\colon pw, \text{MsgHash}\colon H_{m_\text{c}})$.
Let $\varrho_\text{c} = \texttt{encrypt}_{\texttt{ae}}(s, (\text{C}, c, H_r), pk^{\text{ae}}, m'_\text{c})$.
Let $state[c, s, H_r] = 2$.
Send $(m_\text{c}, \varrho_\text{c})$ to A_C.

Approval to send the request message:
$\texttt{processRequestApproval}(m_c, \varrho_c)$
Let $(\text{From}\colon c, \text{To}\colon s, \text{MsgID}\colon H_r, \text{Time}\colon t_c, \text{Body}\colon p_c) = m_c$.
If $|p_c| + |\varrho_c| \geq n[s] - 1$, break.
Let $n[s] = 0$.
Let $t[s] = \texttt{getTime}(s, \mathsf{S})$.
Let $m_c' = \texttt{decrypt}_{\texttt{ae}}(s, \varrho_c)$.
Let $(\text{SecMsgID}\colon r, \text{Pass}\colon pw, \text{MsgHash}\colon H_{m_c}) = m_c'$.
Let $H_r' = \texttt{hash}(r)$.
Let $H_{m_c}' = \texttt{hash}(m_c)$.
If $cor[s] \vee (\texttt{sid}^{\neg\texttt{so}}(s, c, r) \neq \perp)$,
Send (s, c, Test, pw) to SM.
Else,
Send $(s, c, \texttt{sid}^{\neg\texttt{so}}(s, c, r), \text{Test})$ to MX.
Receive $(s, c, \text{Test}, \underline{b})$ from SM.
If $(H_{m_c} \neq H_{m_c}') \vee (H_r \neq H_r') \vee (\neg b) \vee (t_c \leq t_{\min}[s]) \vee (t_c > t[s] + \text{tol}^+[s]) \vee (\exists t', sid_A', c', z'\colon (t', r, sid_A', c', z') \in L[s])$, break.
While $|L[s]| \geq \text{cap}[s]$:
Let $t_{\min}[s] = \min\{t' \mid (t', r', sid_A', c', z') \in L[s]\}$.
For $(t', r', sid_A', c', z') \in L[s]$ with $(\neg z') \wedge (t' \leq t_{\min}[s])$,
Call $\texttt{expire}(s, c', sid_A')$.
Let $L[s] = \{(t', r', sid_A', c', z') \in L[s] \mid t' > t_{\min}[s]\}$.
If $\texttt{sid}^{\neg\texttt{so}}(s, c, r) \neq \perp$,
Let $sid_A = \texttt{sid}^{\neg\texttt{so}}(s, c, r)$.
Let $L[s] = L[s] \cup \{(t, r, sid_A, c, \texttt{false})\}$.
Send $(s, c, sid_A, \text{Request}_{\text{OK}}, p_c)$ to MX.
Else,
Send $(s, \text{Session}, c, cor[s] \vee cor[s, c, H_r], pw, p_c)$ to SM.
Recv $(s, c, \underline{sid_A}, \text{Session})$ from MX.
Let $sessions = sessions \cup (s, c, sid_A, r, \texttt{true})$.
Let $L[s] = L[s] \cup \{(t, r, sid_A, c, \texttt{false})\}$.
Send $(s, c, sid_A, \text{Session}_{\text{OK}}, pw)$ to MX.

Request to send the response message:
$\texttt{processResponseRequest}(s, c, sid_A, p_s)$
Fetch $(s, c, sid_A, \underline{r}, \underline{so})$ from *sessions*.
Fetch $(\underline{t}, r, sid_A, c, \texttt{false})$ from $L[s]$.
Update $(t, r, sid_A, c, \texttt{false})$ in $L[s]$ to $(t, r, sid_A, c, \texttt{true})$.
Let $pk^{\text{sig}} = \texttt{getSigKey}(s, (\mathsf{S}, c, H_r))$.
Let $m_s = (\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon \texttt{hash}(r), \text{Body}\colon p_s)$.
Let $\sigma_s = \texttt{sign}(s, (\mathsf{S}, c, H_r), m_s)$.
Send (m_s, σ_s) to A_{S}.

Approval to send the response message:
$\texttt{processResponseApproval}(m_s, \sigma_s)$
Let $(\text{From}\colon c, \text{To}\colon s, \text{Ref}\colon H_r, \text{Body}\colon p_s) = m_s$.
Fetch $(s, c, \underline{sid_A}, \underline{r}, \texttt{false})$ from *sessions* where $\texttt{hash}(r) = H_r$, or break if no such entry exists.
Let $state[c, s, H_r] = 3$.
If $|p_s| > n[c, s, H_r]$, break.
Let $pk^{\text{sig}} = \texttt{getSigKey}(s, (\mathsf{S}, c, H_r))$.
Let $b = \texttt{verify}(s, (\mathsf{S}, c, H_r), \mathsf{C}, m_s, \sigma_s, pk^{\text{sig}})$.
If $\neg b$, break.
Send $(s, c, sid_A, \text{Response}_{\text{OK}}, p_s)$ to MX.

Reset of the server:
$\texttt{processServerReset}(s)$
For $(t, r, sid_A, c, z) \in L[s]$ with $\neg z$,
Call $\texttt{expire}(s, c, sid_A)$.
Let $t_{\min}[s] = t[s] + \text{tol}^+[s]$ and $L[s] = [\,]$.

Get a value from the random oracle:
`hash`(m)
Send (GetRO, m) to RO.
Recv (RO, $\underline{H_m}$) from RO.
Return H_m.

Get the time of a principal:
`getTime`(pid, sid)
Send $(pid, sid, \text{GetTime})$ to LC.
Recv $(pid, sid, \text{Time}, \underline{t})$ from LC.
Return t.

Get a key from the signature key store:
`getSigKey`(pid, sid)
Send $(pid, sid, \text{GetKey})$ to KS^{sig}.
Recv $(pid, sid, \text{PublicKey}, \underline{pk^{\text{sig}}})$ from KS^{sig}.
Return pk^{sig}.

Get a key from the public key encryption key store:
`getEncKey`$_{\texttt{ae}}(pid, sid)$
Send $(pid, sid, \text{GetKey})$ to KS^{ae}.
Recv $(pid, sid, \text{PublicKey}, \underline{pk^{\text{ae}}})$ from KS^{ae}.
Return pk^{ae}.

Get a signature:
`sign`(pid, sid, m)
Send $(pid, sid, \text{Sign}, m)$ to SIG.
Recv $(pid, sid, \text{Signature}, \underline{\sigma})$ from SIG.
Return σ.

Verify a signature:
`verify`$(pid, sid, ssid, m, \sigma, pk^{\text{sig}})$
Send $(pid, sid, ssid, \text{Init})$ to SIG.
Recv $(pid, sid, ssid, \text{Inited})$ from SIG.
Send $(pid, sid, ssid, \text{Verify}, m, \sigma, pk^{\text{sig}})$ to SIG.
Recv $(pid, sid, ssid, \text{Verified}, \underline{b})$ from SIG.
Return b.

Encrypt a plaintext (public key encryption):
`encrypt`$_{\texttt{ae}}(pid, sid, pk^{\text{ae}}, m)$
Send (pid, sid, Init) to ENC.
Recv $(pid, sid, \text{Inited})$ from ENC.
Send $(pid, sid, \text{Enc}, pk^{\text{ae}}, m)$ to ENC.
Recv $(pid, sid, \text{Ciphertext}, \underline{\varrho})$ from ENC.
Return ϱ.

Decrypt a ciphertext (public key encryption):
`decrypt`$_{\texttt{ae}}(pid, \varrho)$
Send $(pid, \text{Dec}, \varrho)$ to ENC.
Recv $(pid, \text{Plaintext}, m)$ from ENC.
Return m.

Expire a session:
`expire`(s, c, sid_A)
Send $(s, c, sid_A, \text{Expire})$ to MX.
Recv $(s, c, sid_A, \text{Expire}_{\text{OK}})$ from MX.

Corrupt a session:
`corrupt`(s, c, sid_A)
Call `corrupt`(s, c, sid_A, c).
Call `corrupt`(s, c, sid_A, s).

Corrupt a session:
$\texttt{corrupt}(s, c, sid_A, x)$
If $\neg cor[s, c, sid_A, x]$,
Send $(s, c, sid_A, \text{Corrupt}, x)$ to MX.
Recv $(s, c, sid_A, \text{Corrupt}_{\text{OK}}, x)$ from MX.
Let $cor[s, c, sid_A, x] = \texttt{true}$.

Retrieve the nonce of a session:
$\texttt{nonce}(s, c, sid_A)$
Fetch $(s, c, sid_A, \underline{r}, \underline{so})$ from *sessions*.
Return r.

Retrieve the session id's of sessions:
$\texttt{sid}^{\neg\text{so}}(s, c, r)$
Fetch $(s, c, \underline{sid_A}, r, \texttt{false})$ from *sessions*.
Return sid_A.

Retrieve the session id's of sessions:
$\texttt{sids}(s, c, H_r)$
Return $\{sid_A \mid \exists r, \textit{server-only}\colon ((s, c, sid_A, r, \textit{server-only}) \in \textit{sessions}) \wedge (\texttt{hash}(r) = H_r)\}$.

C. IITM's for Mutual Authentication

C.1. Ideal Functionality

C.1.1. Ideal Single-Session Mutual Authentication Functionality $\mathcal{F}_{\text{MA}}^{\text{SS}}$

Tapes: $F_{\text{MA}} \leftarrow\!\!\!\!\leftarrow\!\!\!\rightarrow E_{\text{MA}}, F_{\text{MA}} \dashleftarrow\!\!\dashrightarrow A_{\text{MA}}$

Variables and Initialization:

Variable	Type	Initial Value
$i, j, state$	$\mathbb{N}$	0
n_1, n_2	$\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Authentication: (C.1)
`if` $(\underline{i}, \underline{j})$ is received from E_{MA} while $state = 0$, `do`
Send (i, j) to A_{MA}.
Recv (j, i) from E_{MA}.
Send (j, i) to A_{MA} and let $state = 2$.

Notification 1: (C.2)
`if` (i, j) is received from A_{MA} while $(state = 2) \wedge \neg n_1$, `do`
Send (i, j) to E_{MA} and let $n_1 = \texttt{true}$.

Notification 2: (C.3)
`if` (j, i) is received from A_{MA} while $(state = 2) \wedge \neg n_2$, `do`
Send (j, i) to E_{MA} and let $n_2 = \texttt{true}$.

CheckAddress: After the first message, accept all messages of the form (i, j) or (j, i).

C.1.2. Ideal Multi-Session Mutual Authentication Functionality $\mathcal{F}_{\text{MA}}^{\text{MS}}$

Tapes: $F_{\text{MA}} \leftarrow\!\!\!\!\leftarrow\!\!\!\rightarrow E_{\text{MA}}, F_{\text{MA}} \dashleftarrow\!\!\dashrightarrow A_{\text{MA}}, F_{\text{Out}} \longrightarrow F_{\text{In}}$

Variables and Initialization:

Variable	Type	Initial Value
$i, j, s, t, state$	$\mathbb{N}$	0
n_1, n_2, cor	$\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Initiation: (C.4)
`if` $(\underline{i}, \underline{j}, \underline{s})$ is received from E_{MA} while $state = 0$, `do`
Send (i, j, s) to A_{MA} and let $state = 1$.

Request: (C.5)
`if` $(i, j, s, \underline{t})$ is received from A_{MA} while $state = 1$, `do`
Send (j, i, t, s) to F_{Out}.
Recv (i, j, s) from F_{In}.
Send (i, j, s, t) to A_{MA} and let $state = 2$.

Response: (C.6)
`if` $(i, j, s, \underline{t})$ is received from F_{In} while $state = 1$, `do`
Send (j, i, t) to F_{Out} and let $state = 2$.

Notification: (C.7)
`if` (i, j, s) is received from A_{MA} while $state = 2$, `do`
Send (i, j, s) to E_{MA} and let $state = 3$.

Corruption: $\texttt{Corr}(cor, \texttt{true}, state > 0, \varepsilon, A_{\text{MA}}, \{E_{\text{MA}}\}, E_{\text{MA}}, i, j, s)$

CheckAddress: After the first message, accept all messages that start with $(i, j, s \ldots$.

C.2. Realization

C.2.1. Single-Session Protocol Wrapper $\mathcal{P}_{\Pi}^{\text{SS}}$

Tapes: $F_{\text{MA}} \twoheadleftarrow\!\!\longrightarrow E_{\text{MA}}$, $P_{\Pi} \dashleftarrow\!\dashrightarrow A_{\Pi}$, $P_{\Pi} \longleftarrow\!\!\twoheadrightarrow \mathcal{G}$

Variables and Initialization:

Variable	Type	Initial Value
$i, j, state$	$\mathbb{N}$	0
r	$\{0,1\}^{q(\eta)}$	$r \xleftarrow{R} \{0,1\}^{q(\eta)}$
κ	$listof\{0,1\}^*$	[]
δ, cor	$\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Initiation: (C.8)
`if` $(\underline{i}, \underline{j})$ is received from E_{MA} while $state = 0$, `do`
Send (i, j) to $P_{\mathcal{G}}$.
Recv $(i, j, \underline{a})$ from $P_{\mathcal{G}}$.
Send (i, j) to A_{Π} and let $state = 1$.

Protocol Messages: (C.9)
`if` $(i, j, \underline{m_{\text{in}}})$ is received from A_{Π} while $state \in \{1, 2\}$, `do`
Let $\kappa = \kappa \cdot m_{\text{in}}$.
Let $(\underline{m_{\text{out}}}, \underline{\delta}, \underline{\alpha}) = \Pi(1^{\eta}, i, j, a, \kappa, r)$.
If $\delta = \mathsf{A}$, let $state = 2$.
Send $(i, j, m_{\text{out}}, \delta)$ to A_{Π}.

Notification: (C.10)
`if` (i, j) is received from A_{Π} while $state = 2$, `do`
Send (i, j) to E_{MA} and let $state = 3$.

CheckAddress: After the first message, accept all messages that start with $(i, j, \ldots$.

C.2.2. Multi-Session Protocol Wrapper $\mathcal{P}_{\Pi}^{\text{MS}}$

Tapes: $F_{\text{MA}} \twoheadleftarrow\!\!\longrightarrow E_{\text{MA}}$, $P_{\Pi} \dashleftarrow\!\dashrightarrow A_{\Pi}$, $P_{\Pi} \longleftarrow\!\!\twoheadrightarrow P_{\mathcal{G}}$

Variables and Initialization:

Variable	Type	Initial Value
$i, j, s, state$	$\mathbb{N}$	0
r	$\{0,1\}^{q(\eta)}$	$r \xleftarrow{R} \{0,1\}^{q(\eta)}$
κ	$listof\{0,1\}^*$	[]
δ, cor	$\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Initiation: (C.11)
`if` $(\underline{i}, \underline{j}, \underline{s})$ is received from E_{MA} while $state = 0$, `do`
Send (i, j, s) to $P_{\mathcal{G}}$.
Recv $(i, j, s, \underline{a})$ from $P_{\mathcal{G}}$.
Send (i, j, s) to A_{Π} and let $state = 1$.

Protocol Messages: (C.12)
$\texttt{if}\ (i, j, s, \underline{m_{\text{in}}})$ is received from A_Π while *state* $\in \{1, 2\}$, do
Let $\kappa = \kappa \cdot m_{\text{in}}$.
Let $(\underline{m_{\text{out}}}, \underline{\delta}, \underline{\alpha}) = \Pi(1^\eta, i, j, a, \kappa, r)$.
If $\delta = \mathsf{A}$, let *state* $= 2$.
Send $(i, j, s, m_{\text{out}}, \delta)$ to A_Π.

Notification: (C.13)
$\texttt{if}\ (i, j, s)$ is received from A_Π while *state* $= 2$, do
Send (i, j, s) to E_{MA} and let *state* $= 3$.

Corruption: $\texttt{Corr}(cor, \texttt{true}, state > 0, \varepsilon, A_\Pi, \{E_{\text{MA}}\}, E_{\text{MA}}, i, j, s)$

CheckAddress: After the first message, accept all messages that start with $(i, j, s, \ldots$.

C.2.3. Single-Session Key Generator Wrapper $\mathcal{P}_{\mathcal{G}}^{\text{SS}}$

Tapes: $P_{\mathcal{G}} \leftarrow\!\!\!\rightarrow P_\Pi$, $P_{\mathcal{G}} \dashleftarrow\!\!\dashrightarrow A_{\mathcal{G}}$

Variables and Initialization:

Variable	Type	Initial Value
r	$\{0,1\}^{q(\eta)}$	$r \xleftarrow{R} \{0,1\}^{q(\eta)}$
$sent_{\mathcal{A}}$	$\{\texttt{true}, \texttt{false}\}$	$\texttt{false}$

Steps: `loop`

Key Generation for User: (C.14)
$\texttt{if}\ (\underline{i}, \underline{j})$ is received from P_Π, do
Send $(i, j, \mathcal{G}(1^\eta, i, r))$ to P_Π.

Key Generation for the Adversary: (C.15)
$\texttt{if}\ \mathcal{A}$ is received from $A_{\mathcal{G}}$ while $sent_{\mathcal{A}} = \texttt{false}$, do
Send $\mathcal{G}(1^\eta, \mathcal{A}, r)$ to $A_{\mathcal{G}}$ and let $sent_{\mathcal{A}} = \texttt{true}$.

C.2.4. Multi-Session Key Generator Wrapper $\mathcal{P}_{\mathcal{G}}^{\text{MS}}$

Tapes: $P_{\mathcal{G}} \leftarrow\!\!\!\rightarrow P_\Pi$, $P_{\mathcal{G}} \dashleftarrow\!\!\dashrightarrow A_{\mathcal{G}}$

Variables and Initialization:

Variable	Type	Initial Value
r	$\{0,1\}^{q(\eta)}$	$r \xleftarrow{R} \{0,1\}^{q(\eta)}$
$sent_{\mathcal{A}}$	$\{\texttt{true}, \texttt{false}\}$	$\texttt{false}$

Steps: `loop`

Key Generation for User: (C.16)
$\texttt{if}\ (\underline{i}, \underline{j}, \underline{s})$ is received from P_Π, do
Send $(i, j, s, \mathcal{G}(1^\eta, i, r))$ to P_Π.

Key Generation for the Adversary: (C.17)
$\texttt{if}\ \mathcal{A}$ is received from $A_{\mathcal{G}}$ and $sent_{\mathcal{A}} = \texttt{false}$, do
Send $\mathcal{G}(1^\eta, \mathcal{A}, r)$ to $A_{\mathcal{G}}$ and let $sent_{\mathcal{A}} = \texttt{true}$.

C.3. Simulator $\mathcal{S}_{\mathrm{MA}}^{\mathrm{MS}}$

Tapes: $P_\Pi \leftarrow\!\!\rightarrow A_\Pi$, $P_\mathcal{G} \leftarrow\!\!\rightarrow A_\mathcal{G}$, $A_{\mathrm{MA}} \leftarrow\!\!\rightarrow F_{\mathrm{MA}}$

Variables and Initialization:

Variable	Type	Initial Value
S	set of 4-tuples	$\emptyset$
C	set of 3-tuples	$\emptyset$
$r_\mathcal{G}$	$\{0,1\}^{q(\eta)}$	$r_\mathcal{G} \xleftarrow{R} \{0,1\}^{q(\eta)}$
$sent_\mathcal{A}$	$\{\texttt{true}, \texttt{false}\}$	`false`
a, r	associative array of $\{0,1\}^*$	$\perp$
κ	associative array of lists of $\{0,1\}^*$	$[\,]$
δ, n, cor	associative array of $\{\texttt{true}, \texttt{false}\}$	`false`

Steps: `loop`

Initiation: (C.18)
`if` $(\underline{i}, \underline{j}, \underline{s})$ is received from F_{MA}, `do`
If $(i,j,s) \notin S$,
Let $S = S \cup \{(i,j,s)\}$, $\delta[i,j,s] = \texttt{false}$, $\kappa[i,j,s] = [\,]$, $a[i] = \mathcal{G}(1^\eta, i, r_\mathcal{G})$, $n[i,j,s] = \texttt{false}$
$r[i,j,s] \xleftarrow{R} \{0,1\}^{q(\eta)}$, and $cor[i,j,s] = \texttt{false}$.
Send (i,j,s) to A_Π.

Protocol Messages: (C.19)
`if` $(\underline{i}, \underline{j}, \underline{s}, \underline{m_{\mathrm{in}}})$ is received from A_Π while $((i,j,s) \in S) \wedge \neg cor[i,j,s]$, `do`
Let $\kappa[i,j,s] = \kappa[i,j,s] \cdot m_{\mathrm{in}}$.
Let $(\underline{m_{\mathrm{out}}}, \underline{\delta'}, \underline{\alpha}) = \Pi(1^\eta, i, j, a[i], \kappa[i,j,s], r[i,j,s])$.
If $\delta' = \mathsf{A}$, let $\delta[i,j,s] = \texttt{true}$ and call $\texttt{connect}(i,j,s)$.
Send $(i,j,s,m_{\mathrm{out}},\delta')$ to A_Π.

Notification: (C.20)
`if` $(\underline{i}, \underline{j}, \underline{s})$ is received from A_Π while $\delta[i,j,s] \wedge \neg n[i,j,s]$, `do`
Send (i,j,s) to F_{MA} and let $n[i,j,s] = \texttt{true}$.

Key Generation for the Adversary: (C.21)
`if` $\mathcal{A}$ is received from $A_\mathcal{G}$ while $(S \neq \emptyset) \wedge \neg sent_\mathcal{A}$, `do`
Send $\mathcal{G}(1^\eta, \mathcal{A}, r_\mathcal{G})$ to $A_\mathcal{G}$ and let $sent_\mathcal{A} = \texttt{true}$.

Corruption: (C.22)
`if` $(\underline{i}, \underline{j}, \underline{s}, \text{Corrupt})$ is received from A_Π while $((i,j,s) \in S) \wedge \neg cor[i,j,s]$, `do`
Send $(i,j,s,\text{Corrupt})$ to F_{MA} and let $cor[i,j,s] = \texttt{true}$.
Recv $(i,j,s,\text{Corrupted},m)$ from F_{MA} and send it to A_Π.

Corrupted forward to the adversary: (C.23)
`if` $(\underline{i}, \underline{j}, \underline{s}, \text{Recv}, \underline{m}, \underline{T})$ is received from F_{MA}, `do`
Send (i,j,s,Recv,m,T) to A_Π.

Corrupted forward to the user: (C.24)
`if` $(\underline{i}, \underline{j}, \underline{s}, \text{Send}, \underline{m}, \underline{T})$ is received from A_Π, `do`
Send (i,j,s,Send,m,T) to F_{MA}.

Functions:

Connecting two instances:
$\texttt{connect}(i,j,s)$
If there exists a session id t with $(i,j,s,t) \in C$ or $(j,i,t,s) \in C$, return.
Let t be a session id such that $\kappa[j,i,t]$ is a matching conversation to $\kappa[i,j,s]$.
Send (i,j,s,t) to F_{MA}.
Recv (i,j,s,t) from F_{MA}.
Let $C = C \cup \{(i,j,s,t)\}$.

C.4. Corruption Macro `Corr`

Parameters:

Description	Parameter	Type
Corruption status	*corrupted*	$\{\texttt{true}, \texttt{false}\}$
Is corruption currently possible?	*corruptible*	$\{\texttt{true}, \texttt{false}\}$
Initialization status	*initialized*	$\{\texttt{true}, \texttt{false}\}$
Corruption Message	*corrMsg*	$\{0,1\}^*$
Tape to connect to the adversary	T_{adv}	tape
Tapes controlled by the adversary after corruption	$\mathcal{T}_{\text{user}}$	set of tapes
Tape to connect to the environment	T_{env}	tape
Prefixes for messages to send and receive	$id_1, \ldots, id_n$	$(\{0,1\}^*)^n$

Variables and Initialization:

Variable	Type	Initial Value
res	$\mathbb{N}$	0

Steps: `loop`

Corruption Request: (C.25)
`if` $(id_1, \ldots, id_n, \text{Corrupted?})$ is received from T_{env}, `do`
If *intialized*,
Send $(id_1, \ldots, id_n, corrupted)$ to T_{env}.

Corruption: (C.26)
`if` $(id_1, \ldots, id_n, \text{Corrupt})$ is received from T_{adv}, `do`
If $corruptible \land initalized \land \neg corrupted$:
Let $corrupted = \texttt{true}$.
Send $(id_1, \ldots, id_n, \text{Corrupted}, corrMsg)$ to T_{adv}.

Forward to A (this rule takes precedence over all other rules): (C.27)
`if` $(id_1, \ldots, id_n, \ldots) = \underline{m}$ is received from $T \in \mathcal{T}_{\text{user}}$ while *corrupted*, `do`
Let $res = 0$ and send $(id_1, \ldots, id_n, \text{Recv}, m, T)$ to T_{adv}.

Forward to user: (C.28)
`if` $(id_1, \ldots, id_n, \text{Send}, \underline{m}, \underline{T})$ is received from T_{adv} while
$corrupted \land (T \in \mathcal{T}_{\text{user}}) \land (0 < |m| \leq res) \land (m = (id_1, \ldots, id_n, \ldots))$, `do`
Send m to T.

Resources: (C.29)
`if` $(id_1, \ldots, id_n, \text{Resources}, \underline{r})$ is received from T_{env} while *corrupted*, `do`
Let $res = |r|$ and send $(id_1, \ldots, id_n, \text{Resources}, r)$ to T_{adv}.

CheckAddress: Check for $id_1, \ldots, id_n$.

Bibliography

[ADR02] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. On the security of joint signature and encryption. In Knudsen [Knu02], pages 83–107.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001)*, pages 104–115, 2001.

[AR02] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[ASW98] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. In Kaisa Nyberg, editor, *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 1998)*, volume 1403 of *Lecture Notes in Computer Science*, pages 591–606. Springer, 1998.

[BCC+04] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugène Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. *Web Services Addressing (WS-Addressing)*. World Wide Web Consortium, 2004. `http://www.w3.org/Submission/ws-addressing/`.

[BCJ+06] Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically sound security proofs for basic and public-key Kerberos. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *Lecture Notes in Computer Science*, pages 362–383. Springer, 2006.

[BCK98] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC 1998)*, pages 419–428, 1998.

[BCPQ01] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. Provably authenticated group diffie-hellman key exchange. In Pierangela Samarati, editor, *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS 2001)*, pages 255–264, New York, NY, USA, 2001. ACM.

[BDJR97] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Symposium on Foundations of Computer Science (FOCS 1997)*, pages 394–403. IEEE Computer Society, 1997.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Proceedings of the 18th Annual International Cryptology Conference (CRYPTO 1998)*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.

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

[BF09] Manuel Barbosa and Pooya Farshim. Security analysis of standard authentication and key agreement protocols utilising timestamps. In Bart Preneel, editor, *Proceedings of the Second International Conference on Cryptology in Africa (AFRICACRYPT 2009)*, volume 5580 of *Lecture Notes in Computer Science*, pages 235–253. Springer, 2009.

[BFCZ08] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for TLS. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM, 2008.

[BFG05] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. A semantics for web services authentication. *Theoretical Computer Science*, 340(1):102–153, 2005. Theoretical Foundations of Security Analysis and Design II.

[BFG08] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verifying policy-based web services security. *ACM Transactions on Programming Languages and Systems*, 30(6):1–59, 2008.

[BFGM01] Mihir Bellare, Marc Fischlin, Shafi Goldwasser, and Silvio Micali. Identification protocols secure against reset attacks. In Pfitzmann [Pfi01], pages 495–511.

[BFGO05] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Greg O'Shea. An advisor for web services security policies. In Damiani and Maruyama [DM05], pages 1–9.

[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, *Revised Lectures of the Second International Symposium on Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *Lecture Notes in Computer Science*, pages 197–222. Springer, 2003.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In Guttman [Gut06], pages 139–152.

[BFH+01] Allen Brown, Barbara Fox, Satoshi Hada, Brian LaMacchia, and Hiroshi Maruyama. *SOAP Security Extensions: Digital Signature*. World Wide Web Consortium, 2001. `http://www.w3.org/TR/SOAP-dsig/`.

[BG05] Michael Backes and Thomas Groß. Tailoring the Dolev–Yao abstraction to web services realities. In Damiani and Maruyama [DM05], pages 65–74.

[BH04] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *Proceedings of the 7th International Conference on Information Security (ISC 2004)*, volume 3225 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2004.

[BL06] Michael Backes and Peeter Laud. Computationally sound secrecy proofs by mechanized flow analysis. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 370–379. ACM, 2006.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In George Dinolt, editor, *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96. IEEE Computer Society, 2001.

[Bla07] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF 2007)*, pages 97–111. IEEE Computer Society, 2007.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using diffie-hellman. In Preneel [Pre00], pages 156–171.

[BP03] Michael Backes and Birgit Pfitzmann. A cryptographically sound security proof of the Needham–Schroeder–Lowe public-key protocol. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2003)*, volume 2914 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2003.

[BP04] Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable Dolev–Yao style cryptographic library. In Focardi [Foc04], pages 204–218.

[BP06a] Michael Backes and Birgit Pfitzmann. On the cryptographic key secrecy of the strengthened Yahalom protocol. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, *Proceedings of the IFIP TC-11 21st International Information Security Conference (SEC 2006)*, volume 201 of *IFIP*, pages 233–245. Springer, 2006.

[BP06b] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Dwork [Dwo06], pages 537–554.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Preneel [Pre00], pages 139–155.

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franccois Yergeau. *Extensible Markup Language (XML)*. World Wide Web Consortium, 2008. `http://www.w3.org/TR/REC-xml/`.

[BPW03] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 220–230. ACM, 2003.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Information and Computation*, 205(12):1685–1720, 2007.

[BR93a] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Proceedings of the 13th Annual International Cryptology Conference (CRYPTO 1993)*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1993.

[BR93b] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS 1993)*, pages 62–73, 1993.

[BR95] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: The three party case. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995)*, pages 57–66. ACM, 1995.

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. `http://cseweb.ucsd.edu/~mihir/cse207/classnotes.html`, 2005.

[BWM97] Simon Blake-Wilson and Alfred Menezes. Entity authentication and authenticated key transport protocols employing asymmetric techniques. In Christianson et al. [CCLR98], pages 137–158.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Technical Report 2000/067, Cryptology ePrint Archive, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 136–145. IEEE Computer Society, 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In Focardi [Foc04], pages 219–235.

[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, *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008)*, volume 5201 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2008.

[CCLR98] Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors. *Proceedings of the 5th International Workshop on Security Protocols*, volume 1361 of *Lecture Notes in Computer Science*. Springer, 1998.

[CDL06] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO 2001)*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[CH06] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In Shai

Halevi and Tal Rabin, editors, *Proceedings of the Third Theory of Cryptography Conference (TCC 2006)*, volume 3876 of *Lecture Notes in Computer Science*, pages 380–403. Springer, 2006.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *Proceedings of the 24th Annual International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 2005)*, volume 3494 of *Lecture Notes in Computer Science*, pages 404–421. Springer, 2005.

[Cho07] Kim-Kwang Raymond Choo. A proof of revised Yahalom protocol in the Bellare and Rogaway (1993) model. *The Computer Journal*, 50(5):591–601, 2007.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Pfitzmann [Pfi01], pages 453–474.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Knudsen [Knu02], pages 337–351.

[CKN03] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Dan Boneh, editor, *Proceedings of the 23rd Annual International Cryptology Conference (CRYPTO 2003)*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. Technical Report 2002/140, Cryptology ePrint Archive, 2002.

[CLR07] Yannick Chevalier, Denis Lugiez, and Michaël Rusinowitch. Towards an automatic analysis of web service security. In Boris Konev and Frank Wolter, editors, *Proceedings of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007)*, volume 4720 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2007.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[CTR09] Najah Chridi, Mathieu Turuani, and Michaël Rusinowitch. Decidable analysis for a class of cryptographic group protocols with unbounded lists. In *Computer Security Foundations Symposium* [IEE09], pages 277–289.

[DA06] Tim Dierks and Christopher Allen. *The Transport Layer Security (TLS) Protocol Version 1.1*. Internet Engineering Task Force, 2006. Obsoletes RFC 2246.

[DG04] Giorgio Delzanno and Pierre Ganty. Automatic verification of time sensitive cryptographic protocols. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2004.

[DG06] Ernesto Damiani and Alban Gabillon, editors. *Proceedings of the 3rd Workshop on Secure Web Services (SWS 2006)*. ACM, 2006.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[DKP09] Stéphanie Delaune, Steve Kremer, and Olivier Pereira. Simulation based security in the applied pi calculus. In Ravi Kannan and K. Narayan Kumar, editors, *FSTTCS*, volume 4 of *LIPIcs*, pages 169–180. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[DLMS04] Nancy A. Durgin, Patrick Lincoln, John C. Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[DM04] Ernesto Damiani and Hiroshi Maruyama, editors. *Proceedings of the 2004 Workshop on Secure Web Services (SWS 2004)*. ACM, 2004.

[DM05] Ernesto Damiani and Hiroshi Maruyama, editors. *Proceedings of the 2005 Workshop on Secure Web Services (SWS 2005)*. ACM, 2005.

[DP07] Ernesto Damiani and Seth Proctor, editors. *Proceedings of the 2007 Workshop on Secure Web Services (SWS 2007)*. ACM, 2007.

[DP08] Ernesto Damiani and Seth Proctor, editors. *Proceedings of the 5th ACM Workshop on Secure Web Services (SWS 2008)*. ACM, 2008.

[DPS09] Ernesto Damiani, Seth Proctor, and Anoop Singhal, editors. *Proceedings of the 6th ACM Workshop on Secure Web Services (SWS 2009)*. ACM, 2009.

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

[Dwo06] Cynthia Dwork, editor. *Proceedings of the 26th Annual International Cryptology Conference (CRYPTO 2006)*, volume 4117 of *Lecture Notes in Computer Science*. Springer, 2006.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[ER02] Donald Eastlake and Joseph Reagle. *XML Encryption Syntax and Processing*. World Wide Web Consortium, 2002. `http://www.w3.org/TR/xmlenc-core/`.

[FH10] Dinei Florêncio and Cormac Herley. Where do security policies come from? In Lorrie Faith Cranor, editor, *Proceedings of the 6th Symposium on Usable Privacy and Security (SOUPS 2010)*, New York, NY, USA, 2010. ACM.

[FMCS04] Liang Fang, Samuel Meder, Olivier Chevassut, and Frank Siebenlist. Secure password-based authenticated key exchange for web services. In Damiani and Maruyama [DM04], pages 9–15.

[Foc04] Riccardo Focardi, editor. *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-19 2004)*. IEEE Computer Society, 2004.

[GBN09] M. Choudary Gorantla, Colin Boyd, and Juan Manuel González Nieto. Universally composable contributory group key exchange. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)*, pages 146–156. ACM, 2009.

[GJM99] Juan A. Garay, Markus Jakobsson, and Philip D. MacKenzie. Abuse-free optimistic contract signing. In Michael J. Wiener, editor, *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO 1999)*, volume 1666 of *Lecture Notes in Computer Science*, pages 449–466. Springer, 1999.

[GL06] Oded Goldreich and Yehuda Lindell. Session-key generation using human passwords only. *Journal of Cryptology*, 19(3):241–340, 2006.

[GLS07] Sebastian Gajek, Lijun Liao, and Jörg Schwenk. Breaking and fixing the inline approach. In Damiani and Proctor [DP07], pages 37–43.

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

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[Gol96] Dieter Gollmann. What do we mean by entity authentication? In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P 1996)*, pages 46–54. IEEE Computer Society, 1996.

[Gut06] Joshua Guttman, editor. *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006.

[HB04] Hugo Haas and Allen Brown. *Web Services Glossary*. World Wide Web Consortium, 2004. `http://www.w3.org/TR/ws-gloss/`.

[HE95] Kipp E. B. Hickman and Taher Elgamal. *The SSL Protocol*. Netscape, 1995. `draft-hickman-netscape-ssl-01.txt`.

[HK99] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security*, 2(3):230–268, 1999.

[HMQ04] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *Proceedings of the Third Theory of Cryptography Conference (TCC 2004)*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2004.

[IEE09] IEEE Computer Society. *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, 2009.

[JKL04] Ik Rae Jeong, Jonathan Katz, and Dong Hoon Lee. One-round protocols for two-party authenticated key exchange. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Proceedings of the Second International Conference on Applied Cryptography and Network Security (ACNS 2004)*, volume 3089 of *Lecture Notes in Computer Science*, pages 220–232. Springer, 2004.

[Jou04] Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[Kü06a] Ralf Küsters. Simulation-based security with inexhaustible interactive Turing machines. In Guttman [Gut06], pages 309–320.

[Kü06b] Ralf Küsters. Simulation-based security with inexhaustible interactive Turing machines. Technical Report 2006/151, Cryptology ePrint Archive, 2006.

[KDMR08] Ralf Küsters, Anupam Datta, John C. Mitchell, and Ajith Ramanathan. On the relationships between notions of simulation-based security. *Journal of Cryptology*, 21(4):492–546, 2008.

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

[KMG^{+}07] Anish Karmarkar, Jean-Jacques Moreau, Martin Gudgin, Marc Hadley, Noah Mendelsohn, Yves Lafon, and Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 2: Adjuncts (Second Edition)*. World Wide Web Consortium, 2007. `http://www.w3.org/TR/soap12-part2/`.

[Knu02] Lars R. Knudsen, editor. *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 2002)*, volume 2332 of *Lecture Notes in Computer Science*. Springer, 2002.

[KR06] Eldar Kleiner and A. William Roscoe. On the relationship between web services security and traditional protocols. *Electronic Notes in Theoretical Computer Science*, 155:583–603, 2006.

[KSW09a] Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. Computationally secure two-round authenticated message exchange. Technical Report 2009/262, Cryptology ePrint Archive, 2009.

[KSW09b] Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. A simulation-based treatment of authenticated message exchange. In Anupam Datta, editor, *Proceedings of the 14th Asian Computing Science Conference (ASIAN 2009)*, volume 5913 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2009.

[KSW09c] Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. A simulation-based treatment of authenticated message exchange. Technical Report 2009/368, Cryptology ePrint Archive, 2009.

[KSW10] Klaas Ole Kürtz, Henning Schnoor, and Thomas Wilke. Computationally secure two-round authenticated message exchange. In David Basin and Peng Liu, editors, *Proceedings of the 2010 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2010)*. ACM, 2010.

[KT08a] Ralf Küsters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 270–284. IEEE Computer Society, 2008.

[KT08b] Ralf Küsters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. Technical Report 2008/006, Cryptology ePrint Archive, 2008.

[KT09a] Ralf Küsters and Max Tuengerthal. Universally composable symmetric encryption. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 293–307. IEEE Computer Society, 2009.

[KT09b] Ralf Küsters and Max Tuengerthal. Universally composable symmetric encryption. Technical Report 2009/055, Cryptology ePrint Archive, 2009.

[KT10] Ralf Küsters and Max Tuengerthal. Ideal key derivation and encryption in simulation-based security. Technical Report 2010/295, Cryptology ePrint Archive, 2010.

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Proceedings of the First International Conference on Provable Security (ProvSec 2007)*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

[Low96] Gavin Lowe. Breaking and fixing the Needham–Schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1996)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Luc97] Stefan Lucks. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In Christianson et al. [CCLR98], pages 79–90.

[MA05] Michael McIntosh and Paula Austel. XML Signature element wrapping attacks and countermeasures. In Damiani and Maruyama [DM05], pages 20–27.

[MGMB10] Michael McIntosh, Martin Gudgin, K. Scott Morrison, and Abbie Barbir. *Basic Security Profile Version 1.1*. Web Services Interoperability Organization, 2010. `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html`.

[ML07] Nilo Mitra and Yves Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. World Wide Web Consortium, 2007. `http://www.w3.org/TR/soap12-part0/`.

[MN06] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In Dwork [Dwo06], pages 373–392.

[MSW08] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. In Josef Pieprzyk, editor, *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2008)*, volume 5350 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2008.

[NGG+07a] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. *WS-SecureConversation 1.3*. OASIS Web Services Secure Exchange TC, 2007. `http://www.oasis-open.org/specs/#wssecconv1.3`.

[NGG$^+$07b] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. *WS-SecurityPolicy 1.2*. OASIS Web Services Secure Exchange TC, 2007. http://www.oasis-open.org/specs/#wssecpolv1.2.

[NGG$^+$07c] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. *WS-Trust 1.3*. OASIS Web Services Secure Exchange TC, 2007. http://www.oasis-open.org/specs/#wstrustv1.3.

[NGM$^+$07] Henrik Frystyk Nielsen, Martin Gudgin, Noah Mendelsohn, Marc Hadley, Yves Lafon, Anish Karmarkar, and Jean-Jacques Moreau. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. World Wide Web Consortium, 2007. http://www.w3.org/TR/soap12-part1/.

[NKMHB06] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*. OASIS Web Services Security TC, 2006. http://www.oasis-open.org/specs/#wssv1.1.

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

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. 10.1007/BFb0017309.

[Pfi01] Birgit Pfitzmann, editor. *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 2001)*, volume 2045 of *Lecture Notes in Computer Science*. Springer, 2001.

[Pre00] Bart Preneel, editor. *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT 2000)*, volume 1807 of *Lecture Notes in Computer Science*. Springer, 2000.

[PW01] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P 2001)*, pages 184–201. IEEE Computer Society, 2001.

[RKP09] Alfredo Rial, Markulf Kohlweiss, and Bart Preneel. Universally composable adaptive priced oblivious transfer. In Hovav Shacham and Brent Waters, editors, *Pairing*, volume 5671 of *Lecture Notes in Computer Science*, pages 231–247. Springer, 2009.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and

Willi Meier, editors, *Proceedings of the 11th International Workshop on Fast Software Encryption (FSE 2004)*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[RS09] Phillip Rogaway and Till Stegers. Authentication without elision: Partially specified protocols, associated data, and cryptographic models described by code. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)* [IEE09], pages 26–39.

[RT03] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theoretical Computer Science*, 299(1–3):451–475, 2003.

[SB08] Smriti Kumar Sinha and Azzedine Benameur. A formal solution to rewriting attacks on SOAP messages. In Damiani and Proctor [DP08], pages 53–60.

[SBEW01] Michael Steiner, Peter Buhler, Thomas Eirich, and Michael Waidner. Secure password-based cipher suite for TLS. *ACM Transactions on Information and System Security*, 4(2):134–157, 2001.

[Sho99] Victor Shoup. On formal models for secure key exchange (version 4). Technical Report RZ 3120, IBM, 1999.

[SRE02] David Solo, Joseph Reagle, and Donald Eastlake. *XML Signature Syntax and Processing*. World Wide Web Consortium, 2002. `http://www.w3.org/TR/xmldsig-core/`.

[Sun98] Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification Version 2*. Internet Engineering Task Force, 1998. Obsoletes RFC 1050.

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

[Win99] Dave Winer. *XML-RPC Specification*, 1999.

[Ylö96] Tatu Ylönen. SSH – secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42. USENIX Association, 1996.

Lists of Figures and Tables

List of Figures

2.1. Authentication means and goals 29

3.1. Message flow in four steps 42

4.1. An abstract view of the two systems of IITM's 70
4.2. Legend for illustrations of (systems of) IITM's 71
4.3. The ideal functionality $\mathcal{F}_{\text{S2ME}}$ 73
4.4. An overview of a realization $\mathcal{P}_{\text{S2ME}}$ 82
4.5. A hypothetical approach for a correctness definition 117

5.1. Ideal world and realization for mutual authentication 127

B.1. States and steps of the functionality $\mathcal{F}_{\text{MX}}$ 144

List of Tables

3.1. Input parameters and output values of the algorithms Γ and Σ 43
3.2. The experiment $\text{Exp}_{\Pi,\mathcal{A}}$ 47
3.3. Running times of the procedures of the simulator 63

4.1. Different realizations of $\mathcal{F}_{\text{S2ME}}$ 83