MoSS: Modular Specifications Security Framework

Amir Herzberg[1], Hemi Leibowitz[2], Ewa Syta[3], and Sara Wrótniak[1]

[1] Dept. of Computer Science and Engineering, University of Connecticut, Storrs, CT
[2] Dept. of Computer Science, Bar-Ilan University, Ramat Gan, Israel
[3] Dept. of Computer Science, Trinity College, Hartford, CT**

Abstract. It is hard to prove security for practical cryptographic protocols, which involve assumptions and requirements related to concurrency, delays and synchronization. Current approaches to analyze concurrent computation, e.g., [12], are too complex and do not support computational aspects. On the other hand, cryptographic frameworks such as UC [8] offer only minimal support for concurrency, delays and synchronization. Further, these frameworks use *monolithic specifications*, which are hard to extend to support applied-concurrency. Monolithic specifications have other disadvantages too: they are complex, error-prone and foil reusability and incremental design.

To address these challenges, we present the *Modular Specifications Security (MoSS) framework*, which cleanly separates *requirement specifications* (goals) which a protocol should achieve from *model specifications* (assumptions) under which the protocol is analyzed. This approach allows modular design and analysis of practical cryptographic protocols under different, well-defined notions of delays, synchronization and faults. The resulting specifications and proofs of security are intuitive and reusable.

1 Introduction

For many years, we have come to expect proposals of new cryptographic constructions and schemes to have proofs of security under 'reasonable' assumptions. This expectation is justified as designs based on heuristic and experience, even those proposed by experts, have repeatedly been broken. While assumptions may prove incorrect, the classical cryptographic assumptions, such as hardness of certain problems, so far have proven quite trustworthy. With cryptography becoming critical to the security of networks and systems, the importance of a provable design is hard to dispute.

Indeed, most constructions of cryptographic schemes are proven, as a matter of course, to satisfy rigorous definitions under well-defined and reasonable assumptions. However, in practice, most attacks circumventing the cryptographic

** The work was partially completed during a visiting position at the Dept. of Computer Science and Engineering, University of Connecticut, Storrs, CT

defense, exploit vulnerabilities related to the *concurrency* of *distributed cryptographic protocols*, and not to vulnerabilities of the underlying *cryptographic primitives/functions*. It appears that the protocols using cryptography are not as well analyzed as the underlying cryptographic building blocks, making them more vulnerable and exploitable.

Unfortunately, analysis of concurrent, distributed computation is notoriously challenging, especially for realistic models of delays, synchronization and faults. In the distributed computing community, there exist extensive frameworks allowing analysis of concurrent systems, with non-trivial models of delays and clock-synchronization. Most notable are the Timed I/O Automata [12] and concurrency theory [6]. However, these frameworks are designed for distributed computing and are hard to apply to analysis of cryptographic protocols. In particular, it is hard or impossible to use computational assumptions and different keying and other security assumptions (shared keys, trusted public keys and more), and they do not allow for realistic adversary models. Besides, these frameworks are probably too complex to be applied to the already complex cryptographic protocols. A different approach is taken by *game-based* proofs of security, e.g. [4, 21], which are convenient for capturing the computational aspects and for simple, intuitive definitions and proofs of security. Indeed, game-based definitions and proofs have facilitated the introduction of provable security, and due to its relative simplicity, are still the common way to teach cryptography and are widely used by practitioners. Unfortunately, current game-based definitions do not provide a precise *execution model* defining the *concurrent execution*. This becomes a serious impediment when trying to define and analyze security for real-life goals and environments, often involving partial-synchronization and bounded delay. The MoSS framework, which is presented in this paper, follows the game-based approach, with simple and intuitive definitions and proofs of security, but it does provide a well-defined execution process, and supports concurrency, including different, well-defined notions of delays, synchronization and faults.

Another alternative approach is taken by *simulation-based* frameworks for provable security, including universal composability (UC) [8], its extensions such as iUC, GNUC, IITM and simplified-UC [7, 9, 11, 14, 22], constructive cryptography (CC) [18, 19], and reactive systems [1]. An important feature of these frameworks is their support for *compositions* of provably-secure protocols, allowing *modular design of protocols*. This is a very important property; however, without handling realistic models of concurrency, delays and clock synchronization, we cannot even define or prove security of one (realistic) protocol.

We believe that one challenge in extending simulation-based frameworks to support realistic models of concurrency, communication (delays) and synchronization, is due to the fact that they use *monolithic security specifications*. For example, in UC, security specifications are defined by a monolithic *ideal functionality*. Such monolithic security specifications have two serious drawbacks.

The first drawback is that, unavoidably, **monolithic specifications are more complex** than an equivalent modular specifications. It is hard to *understand, compare, validate and extend* complex (monolithic) definitions and proofs,

as well as to apply automated tools that assist in validation. This is a serious issue: incorrect specifications and incorrect interpretation of specifications may lead to incorrect usage - which is a major cause of vulnerabilities. The complexity of monolithic specifications often also results in complex, hard-to-validate - and sometimes incorrect - proofs. Such concerns were raised repeatedly; Koblitz and Menezes even argue that '...a claimed proof may engender a false sense of security, and may discourage people from further study of the security of a protocol' [13]. Reduction of complexity is also a motivation for iUC, IITM, simplified UC and CC [7,9,14,18,19,22], and GNUC [11], which even claims that there are flaws in the UC's proof of composition, perhaps rooted in its complexity.

The second drawback is that a **monolithic specifications foil reusability and incremental design**. A proof of security under such monolithic specifications cannot be used directly in a proof of an extension of that protocol, designed to address additional goals, or to use more realistic, weaker assumptions. Namely, this prevents *reuse* of definition and analysis efforts from one work to another, when they use different assumptions, e.g., to extend simple protocols and specification, e.g., assuming the simple synchronous 'rounds model', to handle clock-drift or bounded delays.

To address these challenges, we introduce the *Modular Specifications Security (MoSS) framework*. MoSS supports *modular security specifications*, in contrast to the 'classical' approach of monolithic security specifications, using games (experiments) or ideal-functionalities. Instead of having the model and the requirements integrated into monolithic specifications, in MoSS, the model $\mathcal{M}$ and the requirements $\mathcal{R}$ are each defined by an efficiently-computable *predicate*, applied to the *transcript* of the concurrent execution. The concurrent execution and its transcript are well-defined by the *execution process*, a quite simple PPT algorithm specified in Algorithm 1.

MoSS supports different adversarial models, from 'honest-but-curious' to 'proactive' and 'self-stabilizing' faults and more. This only requires definition of simple model predicates and execution operations, and allows reuse of specifications and even results across models. Contrast this with monolithic specifications frameworks, where support for different adversary models is challenging and limited. Similarly, MoSS supports *concrete security*, in a way we consider simple, elegant and precise; concrete security is only partially[4] supported in (the more recent versions) of UC - and in none of the variants mentioned above.

To ease definition and validation of our specifications, we would normally define each model predicate to reflect a single, focused assumption; similarly, we define each requirement predicate to reflect a single, focused requirement. We repeat this until we have all necessary models and requirements. Writing, understanding and validation are easier for such set of focused models and requirements, compared to a single monolithic specification covering all of the assumptions and requirements, e.g., as an ideal functionality. We prove several *modularity and monotonicity lemmas* (Sec. 5), allowing us to combine models and requirements into composite models and requirements.

[4] UC does not support concrete bounds on the adversary or environment runtime.

Model and requirement predicates are easy to *reuse* - in related, and sometimes even unrelated, problems. For example, in this paper we present a simplified instance of an authenticated-broadcast protocol, which assumes - and ensures - bounded delay, and which allows for bounded clock-drift. The bounded-delay and bounded-drift models, are both simple to understand and validate - even without extensive exposure to distributed computing. Furthermore, they are *generic*, in the sense that they would be natural for reuse in specifications and analysis of the many unrelated problems, where bounded-drift and bounded-delay are appropriate.

The use of these separate, focused predicates, also allow *modular protocol development and analysis*. For example, in our analysis of simplified authenticated-broadcast protocol, we first analyze it assuming only a model predicate allowing secure shared-key initialization; this suffice to ensure authenticity but not freshness. Next, we show that by assuming also bounded clock-drift, we can also ensure freshness. Then, we show that by additionally assuming bounded-delay communication, we can ensure bounded delay for the broadcast protocol. The analysis is thus very modular and easy to perform and understand. Compared to proving such properties using monolithic specifications, as when using UC - the analysis using MoSS is like a Lego game!

However, let us clarify that the MoSS framework does not (yet) include a *proof of composition*, as provided by UC and other simulation-based frameworks (see above). This is a serious drawback (shared with other 'game-based' approaches). However, we are quite confident that a composition theorem for MoSS can and will be proven in the future. In this work, we focus on *modularity*, and in particular, *support for cuncurrency, synchronization and realistic delay models*, goals which we find a basic necessity for the analysis of many practical cryptographic protocols - and which are achieved for the first time by MoSS.

It is our hope that MoSS may help to bridge the gap between the theory and practice in cryptography, and facilitate *meaningful* provable security for practical cryptographic protocols and systems.

MoSS is not (yet) supported by *computer-aided security proof tools*, such as [3, 5, 20]. Such tools use formal, machine-checkable approaches, with the promise of easier, less error prone and automatically generated or verified proofs. While not without flaws (see [2] for a comprehensive discussion), we view such an automated approach to proofs as synergistic with MoSS but we leave it to future work to explore adding automatic verification capabilities.

Example: Provably-secure X.509 PKI. Public Key Infrastructure (PKI) schemes amply illustrate the above challenges in practice. PKI is a crucial component of applied cryptography. Current PKI systems are mostly based on the X.509 standard [10], but there are many other proposals; the most significant is Certificate Transparency (CT) [15], which adds significant goals and cryptographic mechanisms. These realistic PKI systems have non-trivial requirements; in particular, synchronization is highly relevant, to deal with such basic aspects as revocation (and beyond). Furthermore, while the basic X.509 design is quite

simple, more advanced PKIs (such as CT) are non-trivial, and definitely require precise definitions and analysis.

Recently, [16] presented the first rigorous study of practical[5] PKI schemes by using MoSS. Specifically, they define models and requirements for practical PKI schemes and prove security of the X.509v2 PKI scheme. The analysis in [16] reuse our bounded-delay and bounded-drift models; similarly, follow-up work is expected to use the models and requirement predicates defined in [16], to prove security for additional PKI schemes, e.g., the important *certificate transparency* PKI, being standardized by the IETF [15].

Organization. Section 2 introduces **Exec**, the adversary-driven execution process. Sections 3 and 4 present the model and requirement specifications, respectively. Section 5 presents the modularity lemmas. Section 7 explains how MoSS supports concrete security. Section 8 demonstrates how to apply our framework using a simplified authenticated broadcast protocol. We conclude and discuss future work in Section 9.

2 Execution Process

In this section, we present the generic *adversary-driven execution process*, a key component of MoSS, defining the execution of a given protocol $\mathcal{P}$ 'controlled' by a given adversary $\mathcal{A}$. MoSS separates the execution process from the model $\mathcal{M}$ under which the protocol is analyzed and the requirements $\mathcal{R}$ defining $\mathcal{P}$'s goals, both of which are described in the following two sections. This separation allows different model assumptions using the same execution process, simplifying the analysis and allowing reusability of definitions and results.

2.1 $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$: An Adversary-Driven Execution Process

The execution process $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$, as defined by the pseudo-code in Algorithm 1 and illustrated in Figures 1-2, specifies the details of running a given protocol $\mathcal{P}$ with a given adversary $\mathcal{A}$, both modeled as efficient (PPT) functions, given parameters $params$. The parameters consists of two subsets: the adversary's parameters $params.\mathcal{A}$ and the protocol's parameters $params.\mathcal{P}$. Note that the model $\mathcal{M}$ is not an input to the execution process; it is only applied to the transcript T of the protocol run produced by $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$, to decide if the adversary adhered to the model, in effect restricting the adversary's capabilities. $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$ allows the adversary to have an *extensive control* over the execution; the adversary decides, at any point, which entity is invoked next, with what operation and with what inputs.

Notation. To allow the execution process to apply to protocols with multiple functions and operations, we define the entire protocol $\mathcal{P}$ as a *single* PPT algorithm and use parameters to specify the exact operations and their inputs.

[5] Grossly-simplified PKI ideal functionalities were studied, e.g., in [11], but without considering even basic aspects such as revocation and expiration.

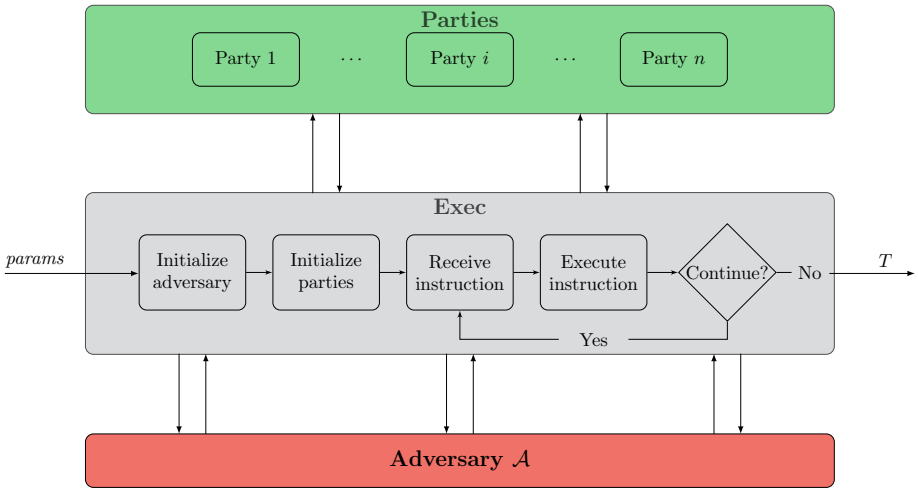

Fig. 1: High level overview of MoSS's execution process.

Specifically, to invoke an operation defined by $\mathcal{P}$ over some entity i, we use the following notation: $\mathcal{P}[opr](s, inp, clk)$, where opr identifies the specific 'operation' or 'function' to be invoked, s is the *local state* of entity i, inp is the set of inputs to opr, and clk is the value of the local clock of entity i. The output of such execution is a tuple (s', out), where s' is the state of entity i *after* the operation is executed and out is the output of the executed operation, which is made available to the adversary. We refer to $\mathcal{P}$ as 'algorithms' (in PPT) but formally they are mappings from strings to algorithms; this should be interpreted as a 'master algorithm' which accepts the 'label' and calls the appropriate 'subroutine' so that the mapping from strings to PPT algorithms is equivalent to a single PPT algorithm.

We use *index notation* to refer to cells of 'arrays'. For example, $out[e]$ refers to the value of the e^{th} entry of the array out. Specifically, e represents the index (counter) of execution events. Note that e is *never* given to the protocol; every individual entity has a separate state, and may count the events that *it* is involved in, but if there is more than one entity, an entity cannot know the current value of e - it is *not* a clock and is not controlled by the adversary. Clocks and time are handled differently, as we now explain.

In every invocation of the protocol, one of the inputs set by the adversary is referred to as the *local clock* and denoted clk. In addition, in every event, the adversary defines a value τ which we refer to as the *real time clock*. Thus, to refer to the local clock value and the real time clock value of event e, the execution process uses $clk[e]$ and $\tau[e]$, respectively. Both clk and τ are included in the transcript T; this allows a model predicate to enforce different *synchronization models/assumptions* - or not to enforce any, which implies a completely asynchronous model.

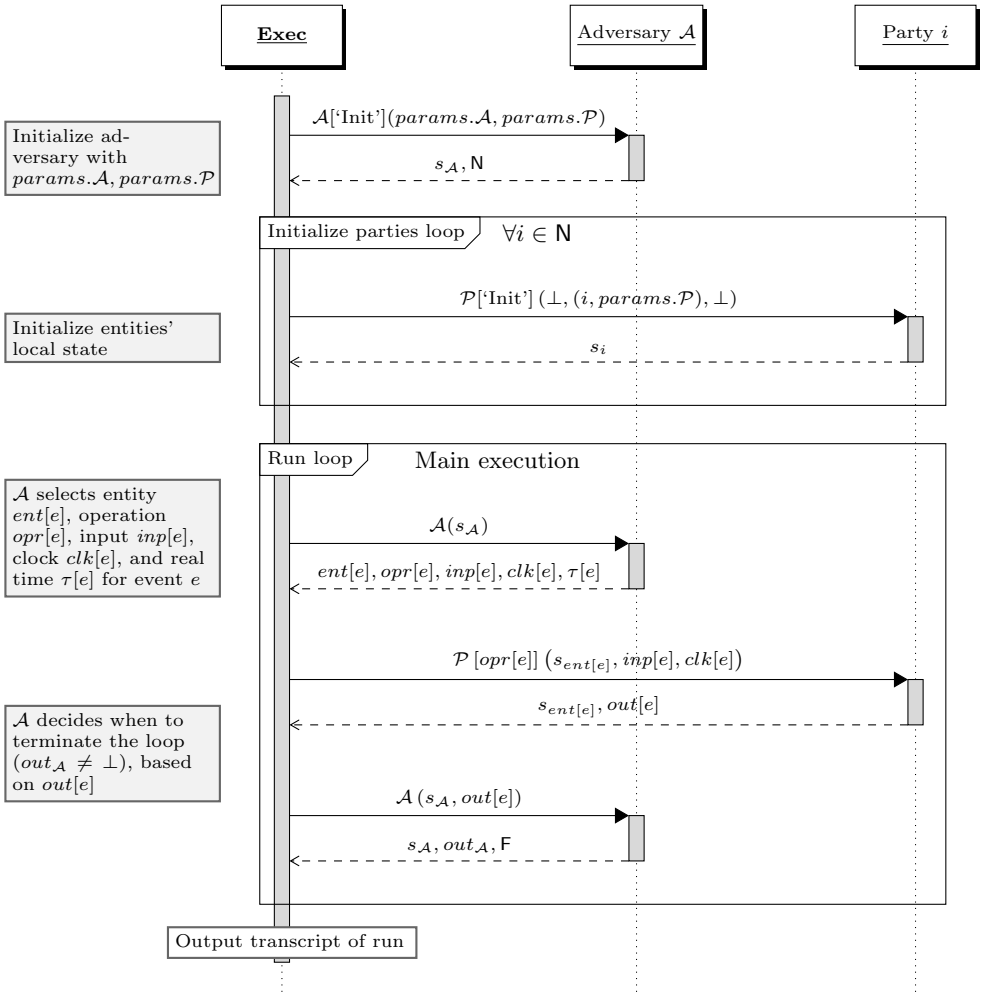

Fig. 2: Illustrative description of MoSS's execution process.

Algorithm 1 Adversary-Driven Execution Process $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$

1: $(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}(\text{'Init'})(params.\mathcal{A}, params.\mathcal{P})$ ▷ *Initialize adversary with* $params.\mathcal{A}, params.\mathcal{P}$

2: $\forall i \in \mathsf{N}: \; s_i \leftarrow \mathcal{P}[\text{'Init'}]\,(\perp, (i, params.\mathcal{P}), \perp)$ ▷ *Initialize entities' local state*

3: $e \leftarrow 0$ ▷ *Initialize loop's counter*

4: **repeat**

5: $\quad e \leftarrow e + 1$ ▷ *Advance the loop counter*

6: $\quad (ent[e], opr[e], inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$ ▷ *$\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event e*

7: $\quad (s_{ent[e]}, out[e]) \leftarrow \mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$

8: $\quad (s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}\,(s_{\mathcal{A}}, out[e])$ ▷ *$\mathcal{A}$ decides when to terminate the loop ($out_{\mathcal{A}} \neq \perp$), based on $out[e]$*

9: **until** $out_{\mathcal{A}} \neq \perp$

10: $T \leftarrow (\, out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot]\,)$

11: Return T ▷ *Output transcript of run*

Construction. The execution process (Algorithm 1) consists of three main components: the initialization, main execution loop and termination.

Initialization (lines 1-3). In line 1, we allow the adversary to set their state $s_{\mathcal{A}}$ and to choose the set of entities N; note that we initialize the adversary with its own parameters $params.\mathcal{A}$ as well as with the protocol's $params.\mathcal{P}$. In line 2, we set the initial state s_i for each entity i by invoking the protocol-specific 'Init' operation with inputs $(i, params.\mathcal{P})$, where each entity receives its identifier i and the security parameters $params.\mathcal{P}$ and performs its initialization operation; note that this implies a convention where protocols are initialized by this operation - all other operations are up to the specific protocol. The reasoning behind such convention is that initialization is an extremely common operation in many protocols; that said, protocols without initialization can use an empty 'Init' operation and protocols with complex initialization process can use other operations defined in $\mathcal{P}$ in the main execution loop (lines 4-9), to implement initialization process which cannot be performed via single 'Init' call. In line 3, we initialize e, which we use to index the events of the execution, i.e., e is incremented by one (line 5) each time we complete one 'execution loop' (lines 4-9).

Main execution loop (lines 4-9). The execution process affords the adversary $\mathcal{A}$ extensive control over the execution. Specifically, in each event e, $\mathcal{A}$ determines (line 6) an operation $opr[e]$, along with its inputs, to be invoked by an entity $ent[e] \in \mathsf{N}$. The adversary also selects $\tau[e]$, the global, real time clock value. Afterwards, the event is executed (line 7).

In line 8, the adversary processes the output $out[e]$ of the operation $opr[e]$. The adversary may modify its state $s_{\mathcal{A}}$, and outputs a value $out_{\mathcal{A}}$; when $out_{\mathcal{A}} \neq \perp$, the execution moves to the termination phase; otherwise the loop continues.

Termination (lines 10-11). Upon termination, the process returns the *execution transcript* T (line 11), containing the relevant values from the execution. Namely, T contains the adversary's output $out_{\mathcal{A}}$, the index of the last event e, the set of entities N, and the set of faulty entities F (produced in line 8) as well as the values of $ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot]$ and $out[\cdot]$ for all invoked events. We allow $\mathcal{A}$ to output F to accommodate different fault modes, i.e., an adversary model can specify which entities are included in F (considered 'faulty') which then can be validated using an appropriate model.

2.2 Extending the Execution Process

In Section 2.1, we described the design of the $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$ execution process, presenting a generic execution process which imposes only basic limitations. We now describe the $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process, an extension of $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$, which provides flexibility in accommodating many different models and supports various applications in a 'built-in' manner, without the need for any further modifications to the execution process, and yet, maintains the appealing simplicity of $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$. That said, a key advantage of MoSS is that the execution process can still be further extended to support custom complex properties; we demonstrate such extension in Section 6.

The $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process, as defined by the pseudo-code in Algorithm 2, specifies the details of running a given protocol $\mathcal{P}$ with a given adversary $\mathcal{A}$, both modeled as efficient (PPT) functions, given *a specific set of execution operations* $\mathcal{X}$ and parameters $params$. The set $\mathcal{X}$ is a specific *set of extra operations* through which the execution process provides built-in yet flexible support for various adversarial capabilities. For example, the set $\mathcal{X}$ can contain functions which allow the adversary to perform specific functionally on one of the entities, functionality which the adversary cannot achieve via the execution of $\mathcal{P}$. We detail and provide concrete examples of such functionalities in Sec. 2.3.

Changes to the $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$ execution process. The input parameters consists of three subsets: the adversary's parameters $params.\mathcal{A}$, the protocol's parameters $params.\mathcal{P}$ and the execution process's parameters $params.\mathcal{X}$. In addition to the extensive control the adversary had over the execution, the adversary now can decides not only which entity is invoked next, but also whether the operation is from the set $\mathcal{X}$ of execution operations, or from the set of operations supported by $\mathcal{P}$; while we did not explicitly write it, some default values are returned if the adversary specifies an operation which does not exist in the corresponding set.

To invoke an operation defined by $\mathcal{P}$ over some entity i, we use the same notation as before, but the output of such execution contains an additional output value *sec-out*, where *sec-out*$[e][\cdot]$ is a 'secure output' - namely, it contains values that are shared only with the execution process itself, and *not shared with the adversary*; e.g., such values may be used, if there is an appropriate operation in $\mathcal{X}$, to establish a 'secure channel' between parties, which is not visible to $\mathcal{A}$. In *sec-out*, the first parameter denotes the specific event e in which the secure output was set; the second one is optional, e.g., may specify the 'destination' of

the secure output. Similarly, $\mathcal{X}$ is also defined as a single PPT algorithm and we use a similar notation to invoke its operations: $\mathcal{X}[opr](s_{\mathcal{X}}, s, inp, clk, ent)$, where opr, s, inp, clk are as before, and $s_{\mathcal{X}}$ is the execution process's state and ent is an entity identifier.

Algorithm 2 Adversary-Driven Execution Process $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$

1: $(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}[\text{'Init'}](params.\mathcal{A}, params.\mathcal{P})$ ▷ *Initialize adversary with* $params.\mathcal{A}, params.\mathcal{P}$

2: $\forall i \in \mathsf{N}: \; s_i \leftarrow \mathcal{P}[\text{'Init'}]\,(\perp, (i, params.\mathcal{P}), \perp)$ ▷ *Initialize entities' local state*

3: $\boldsymbol{s_{\mathcal{X}} \leftarrow params}$ ▷ ***Initial exec state***

4: $e \leftarrow 0$ ▷ *Initialize loop's counter*

5: **repeat**

6: $\quad e \leftarrow e + 1$ ▷ *Advance the loop counter*

7: $\quad (ent[e], opr[e], \mathbf{type[e]}, inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$ ▷ *$\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event e*

8: $\quad$ **if** $\boldsymbol{type[e] = \text{'}\mathcal{X}\text{'}}$ **then** ▷ **If $\mathcal{A}$ chose to invoke an operation from $\mathcal{X}$.**

9: $\quad\quad \boldsymbol{(s_{\mathcal{X}}, s_{ent[e]}, out[e],}$ *sec-out*$\boldsymbol{[e][\cdot])} \leftarrow \boldsymbol{\mathcal{X}\,[opr[e]]\,(s_{\mathcal{X}}, s_{ent[e]}, inp[e], clk[e], ent[e])}$

10: $\quad$ **else** ▷ **$\mathcal{A}$ chose to invoke an operation from $\mathcal{P}$.**

11: $\quad\quad (s_{ent[e]}, out[e],$ *sec-out*$[e][\cdot]) \leftarrow \mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$

12: $\quad$ **end if**

13: $\quad (s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}\,(s_{\mathcal{A}}, out[e])$ ▷ *$\mathcal{A}$ decides when to terminate the loop ($out_{\mathcal{A}} \neq \perp$), based on $out[e]$*

14: **until** $out_{\mathcal{A}} \neq \perp$

15: $T \leftarrow (\, out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], type[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot],$ *sec-out*$[\cdot][\cdot]\,)$

16: Return T ▷ *Output transcript of run*

Construction. The extended execution process (Algorithm 2) consists of the following modifications. The initialization phase (lines 1-4) has an additional line (line 3), where we initialize the 'execution operations state' $s_{\mathcal{X}}$ to the set of all parameters $params$; this state is used by execution operations (in $\mathcal{X}$), allowing them to be defined as (stateless) functions. The rest of the initialization lines are the same.

The main execution loop (lines 5-14) is as before, but with one difference, where the adversary $\mathcal{A}$ determines on line 7 the type of operation $type[e]$ to be invoked by an entity $ent[e] \in \mathsf{N}$. The operation type $type[e] \in \{\text{'}\mathcal{X}\text{'}, \text{'}\mathcal{P}\text{'}\}$ indicates if the operation $opr[e]$ is protocol-specific (defined in $\mathcal{P}$) or is it one of the execution process operations (defined in $\mathcal{X}$). Afterwards, the event is executed (lines 8-11) based on the operation type, either '$\mathcal{X}$', if specified, or '$\mathcal{P}$', otherwise.

The termination phase (lines 15-16) is the same as before, but also include in the transcript the *type*[·] values and the *sec-out*[·][·] for all invoked events. We allow $\mathcal{A}$ to output F to accommodate different fault modes, i.e., an adversary model can specify which entities are included in F (considered 'faulty') which then can be validated using an appropriate model. Private values, such as entities' private keys, are not part of the execution transcript unless they were explicitly included in the output due to an invocation of an operation from $\mathcal{X}$ that would allow it.

2.3 Using $\mathcal{X}$ to Define Specification and Entity-Faults Operations

The 'default' execution process is defined by an empty $\mathcal{X}$ set. This provides the adversary $\mathcal{A}$ with *Man-in-the-Middle (MitM)* capabilities, and even beyond: $\mathcal{A}$ receives all outputs, including messages sent, and controls all inputs, including messages received; furthermore, $\mathcal{A}$ controls the values of the local clocks. A non-empty set $\mathcal{X}$ can be used to define *entity-fault operations* and *specification operations*; let us discuss each of these two types of execution-process operations.

Specification-operations. Some model and requirement specifications, require a special execution-process operation, possibly involving some information which must be kept private from the adversary. One example are *indistinguishability* requirements, which are defined in Sec. 4.3.1 using three operations in $\mathcal{X}$: 'Flip', 'Challenge' and 'Guess', whose meaning most readers can guess (and confirm the guess in Sec. 4.3.1).

The 'Sec-in' $\mathcal{X}$-operation. As a simple example of a useful specification operation, we now define the *'Sec-in'* operation, which allows the execution process to provide a secure input from one entity to another, *bypassing* the adversary's MitM capabilities. This operation can be used for different purposes, such as to assume secure shared-key initialization - for example, see Sec. 8.2.1.

$$\mathcal{X}[\text{'Sec-in'}](s_{\mathcal{X}}, s, e', clk, ent) \equiv [s_{\mathcal{X}} || \mathcal{P}[\text{'Sec-in'}](s, sec\text{-}out[e'][ent], clk)] \quad (1)$$

As can be seen, invocation of the 'Sec-in' operation returns the state $s_{\mathcal{X}}$ unchanged (and unused); the other outputs are simply defined by invoking the 'Sec-in' operation of the protocol $\mathcal{P}$, with input *sec-out*$[e'][ent]$ - the *sec-out* output of the event e' intended for entity *ent*.

Note, that although 'Sec-in' facilitates delivery of data from some entity to another while ensuring that the adversary is unable to access this data, it does not provide authentication, namely, the receiving entity cannot rely on the authenticity of the inputted data.

Entity-fault operations. It is quite easy to define $\mathcal{X}$-operations that facilitate different types of entity-fault models, such as *honest-but-curious, byzantine (malicious), adaptive, proactive, self-stabilizing, fail-stop* and others. Let us give informal examples of three fault operations:

'Get-state': provides $\mathcal{A}$ with the entire state of the entity. Assuming no other entity-fault operation, this is the 'honest-but-curious' adversary; note that the adversary may invoke 'Get-state' after each time it invokes the entity, to know its state all the time.

'Set-output': allows $\mathcal{A}$ to force the entity to output specific values. A 'Byzantine' adversary would use this operation whenever it wanted the entity to produce specific output.

'Set-state': allows $\mathcal{A}$ to set any state to an entity. For example, the 'self-stabilization' model amounts to an adversary that may perform a 'Set-state' for every entity (once, at the beginning of the execution).

Comments. Defining these aspects of the execution in $\mathcal{X}$, rather than having a particular choice enforced as part of the execution process, provides significant flexibility and makes for a simpler execution process.

Note that even when the set $\mathcal{X}$ is non-empty, i.e., contains some non-default operations, the adversary's *use* of these operations may yet be restricted for the adversary to satisfy a relevant *model*. We present models specifications in Sec. 3.

The operations in $\mathcal{X}$ are defined as (stateless) *functions*. However, the execution process provides *state* $s_{\mathcal{X}}$ that these operations may use to store values across invocations; the same state variable may be used by different operations. For example, the 'Flip', 'Challenge' and 'Guess' $\mathcal{X}$-operations, used to define *indistinguishability* requirements in Sec. 4.3.1, use $s_{\mathcal{X}}$ to share the value of the bit flipped (by the 'Flip' operation).

3 Model Specifications

The execution process, described in Sec. 2, specifies the details of running a protocol $\mathcal{P}$ against an adversary $\mathcal{A}$ which has an extensive control over the execution. In this section, we present the next component of MoSS: the model predicate π, applied to the execution-transcript T. Intuitively, adversary $\mathcal{A}$ *satisfies model predicate* π, if for (almost) all execution-transcripts T of $\mathcal{A}$, predicate π holds i.e.: $\pi(T, params) = \text{TRUE}$, where $params$ are the parameters used in the execution process (Sec. 3.1). One may say that the model ensures that the great power of the adversary over the execution is used 'with great responsibility'.

The separation between the execution process and the model predicates has significant advantages to the development and analysis of secure protocols. First, it allows us to use the same - relatively simple - execution process, for the analysis of many different protocols. Second, it makes it easy to define multiple simple models, each focusing on a different assumption or restriction, and require that the adversary satisfy all of them - by simple conjunction.

The model captures all of the assumptions regarding the environment and the capabilities of the adversary, including aspects typically covered by (often informal) *communication model, synchronization model* and *adversary model*:

Adversary capabilities: The adversary capabilities (often referred to as the adversary model), define the computational resources of the adversary, e.g., probabilistic polynomial time (PPT), as well as other capabilities, e.g., ciphertext only (CTO) and chosen ciphertext attacks (CCA) capabilities.

Communication assumptions: The communication assumptions define the properties of the underlying communication mechanism, such as reliable/unreliable

communication, FIFO or non-FIFO, authenticated or not, bounded delay, fixed delay or asynchronous, and so on.

Synchronization assumptions: The synchronization assumptions define the availability and properties of per-entity clocks. Common models include purely asynchronous clocks (no synchronization), bounded-drift clocks, and synchronized clocks.

In Sec. 3.1, we define the notion of a *specification* - which we use to define both models and requirements. In Sec. 3.2, we define the notion of a *model-satisfying adversary*. Finally, in Sec. 3.3, we give an example of a model specification.

3.1 Specifications

We first define a general notion of a specification, which we use both for (adversary, communication and synchronization) *models* and for *requirements* (from a given protocol).

We define a specification ξ as a pair, $\xi = (\pi, \beta)$, where π is the *specification predicate* and β is the *base function*. A specification predicate takes as input an execution transcript T and parameters $params$; when $\pi(T, params) = \top$, we say that execution satisfies the predicate π for the given value of $params$. The base function gives the 'base' probability of success for an adversary. For many integrity specifications, e.g. forgery, the base function is 2^{-l}, where l is the output block size; and for indistinguishability-based specifications (see Sec. 4.3), the base function is often $\frac{1}{2}$.

We next define the advantage of adversary $\mathcal{A} \in PPT$ against protocol $\mathcal{P}$ for specification $\xi = (\pi, \beta)$ and the execution-operations set $\mathcal{X}$, as a function of the parameters $params$. This is the probability that $\pi(T, params) = \perp$, for the transcript T of a random execution: $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$.

Definition 1 (Advantage of $\mathcal{A}$ against $\mathcal{P}$ for specification ξ and execution-operations set $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X} \in PPT$ and let $\xi = (\pi, \beta)$ be a specification. The advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification ξ and execution-operations set $\mathcal{X}$ is defined as:*

$$\epsilon^{\xi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \stackrel{def}{=} \max\left\{0, \Pr\left[\begin{array}{c}\pi(T) = \perp, \ where \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{array}\right] - \beta(params)\right\} \quad (2)$$

3.2 Model-Satisfying Adversary

Models are used to restrict the capabilities of the adversary as well as the events that can happen in the execution process. This includes limiting of the possible faults, defining initialization assumptions, and defining the communication and synchronization models. Hence, we ensure that a given adversary $\mathcal{A}$ followed the restrictions of a given model $\mathcal{M} = (\pi, \beta)$ in a given execution transcript T by applying π to T and $params$. Next, we formally define what it means for adversary $\mathcal{A}$ to *satisfy $\mathcal{M}$ with negligible advantage*.

Definition 2 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with negligible advantage using execution operations $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{X} \in PPT$, and let $\mathcal{M} = (\pi, \beta)$ be a model specification. We say that adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with negligible advantage using execution operations $\mathcal{X}$, denoted as $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}$, if for every protocol $\mathcal{P} \in PPT$ and params $\in \{0,1\}^*$, where params includes a unary string params.$\mathcal{P}$.1^κ whose length is at least half[6] of the total params (i.e., $\|params\| \leq 2 \cdot 1^\kappa$), the advantage of $\mathcal{A}$ against $\mathcal{P}$ for $\mathcal{M}$ and $\mathcal{X}$ is negligible in params.$\mathcal{P}$.1^κ, i.e.: $\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)$.*

3.3 Example: the Bounded-Clock-Drift Model $\mathcal{M}^{\mathbf{Drift}}_{\boldsymbol{\Delta_{clk}}}$

To demonstrate a definition of a model predicate, we present the $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ model, defined as $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}} = (\pi^{\text{Drift}}_{\Delta_{clk}}, \beta^{\text{Drift}}_{\Delta_{clk}})$, where $\beta^{\text{Drift}}_{\Delta_{clk}} = 0$ (i.e., the base function is always zero, like for most integrity properties). The predicate $\pi^{\text{Drift}}_{\Delta_{clk}}$ bounds the clock drift, by enforcing two restrictions on the execution: (1) each local-clock value ($clk[\hat{e}]$) must be within Δ_{clk} drift from the real time $\tau[\hat{e}]$, and (2) the real time values should be monotonically increasing. As a special case, when $\Delta_{clk} = 0$, this predicate corresponds to a model where the local clocks are fully synchronized, i.e., there is no difference between entities' clocks. See Algorithm 3.

Algorithm 3 $\pi^{\text{Drift}}_{\Delta_{clk}}$ (T) Predicate

```
1: return (
2:     ∀ê ∈ {1, ..., T.e}:                          ▷ For each event
3:         |T.clk[ê] − T.τ[ê]| ≤ Δ_clk              ▷ Local clock is within Δ_clk drift from real time
4:         and if ê ≥ 2 then T.τ[ê] ≥ T.τ[ê − 1]     ▷ And in each consecutive events the real time difference is monotonically increasing
    )
```

4 Requirement Specifications

To complete the final piece of the MoSS framework, we now define and discuss *requirement* specifications. A requirement is a pair $\mathcal{R} = (\pi, \beta)$ of a predicate (π) and base function (β); it defines a property that a protocol aims to achieve, e.g., security, correctness or liveness requirements.

Combining the three components of the framework, the execution process, model specifications and requirement specifications, brings about the modularity

[6] The reason for the $\|params\| \leq 2 \cdot 1^\kappa$ restriction is to prevent a situation where the adversary receives an input parameter which is much longer than κ bits, e.g., input whose length is exponential in κ, allowing the adversary to run in time which is exponential in the security parameter and thereby 'break' the cryptographic assumptions, e.g., expose secret keys by exhaustive search.

and flexibility of the MoSS framework, and allows to easily analyze whether a given protocol $\mathcal{P}$ ensures a specific requirement $\mathcal{R}$, or a set of requirements, under given model $\mathcal{M}$, interacting with any PPT adversary $\mathcal{A}$.

Namely, different works may reuse the same requirements (and execution process) but use other, possibly more realistic (and more complex) models, expressing different adversary capabilities, restrictions on usage, and assumptions on communication and synchronization. Similarly, different works may reuse the same models to study additional requirements.

The separation between the definition of the model and of the requirements also allows definition of *generic requirement predicates.*, which are applicable to different protocols regardless of their specific functionalities. We identify four generic requirement predicates that appear relevant to many security protocols. These requirement predicates focus on attributes of messages, i.e., non-repudiation, and on detection of misbehaving entities (see Appendix B.2).

Our approach is quite different from the current way of defining requirements for cryptographic schemes and protocols. While it does take some time and effort to get used to the separate models and requirements, we found that with a bit of practice, the advantages become clear and the approach becomes natural and convenient, facilitating modularity and reuse of models and requirements.

4.1 Model-Secure Requirements

A protocol $\mathcal{P}$ would typically have multiple properties, i.e., satisfy multiple requirements. Let $\mathcal{R} = (\pi, \beta)$ be a requirement. Let b be the outcome of π applied to (T), where T is a transcript of the execution process $(T = \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params))$ and $params$ are the parameters, i.e., $b \leftarrow \pi(T)$; if $b = \perp$ then we say that *requirement predicate π was not satisfied* in the execution of $\mathcal{P}$, or that the *adversary won* in this execution. If $b = \top$, then we say that *requirement predicate π was satisfied* in this execution, or that the *adversary lost.*

Next, we formally define what it means for $\mathcal{P}$ to *satisfy $\mathcal{R}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$.*

Definition 3 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$). *Let $\mathcal{X} \in PPT$, and let $\mathcal{R} = (\pi, \beta)$ be a requirement specification. We say that* protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$*, denoted $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R}$, if for every PPT adversary $\mathcal{A}$ that satisfies $\mathcal{M}$ with negligible advantage and parameters $params \in \{0,1\}^*$, where $params$ includes a unary parameter $params.\mathcal{P}.1^\kappa$ whose length is at least half of the total params (i.e., $\|params\| \leq 2 \cdot 1^\kappa$), the advantage of $\mathcal{A}$ against $\mathcal{P}$ for $\mathcal{R}$ and $\mathcal{X}$ is negligible in $params.\mathcal{P}.1^\kappa$, i.e:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R} \stackrel{def}{=} \left(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{poly} \mathcal{M}\right) : \epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa) \tag{3}$$

4.2 $\pi_{\mathsf{AuthComRcv}_{\Delta}}$: Freshness and Authenticity Requirement Predicate

To demonstrate a definition of a security specification predicate, we present the $\pi_{\mathsf{AuthComRcv}_{\Delta}}$ predicate, which ensures that in a given protocol only broadcast messages are received (authentication) and only if they were sent within the last Δ real time (freshness). The $\pi_{\mathsf{AuthComRcv}_{\Delta}}$ requirement predicate is shown in Algorithm 4.

Algorithm 4 $\pi_{\mathsf{AuthComRcv}_{\Delta}}(T)$ Predicate

```
1: return (
2:   ∀ê_R ∈ {1, ..., T.e}:
3:     if T.out[ê_R] = ('Receive', m):          ▷ For each event where a message is received
4:     ∃ê_B ∈ {1, ..., ê_R − 1}                 ▷ There is a previous event
5:       s.t. T.opr[ê_B] = 'Broadcast':         ▷ Which was a 'Broadcast' event
6:       and T.τ[ê_R] − T.τ[ê_B] ≤ Δ            ▷ Within the last Δ real time
7:       and T.inp[ê_B].m = m                   ▷ Where the input message was m
   )
```

4.3 Supporting Confidentiality and Indistinguishability

The MoSS framework is capable of supporting various security goals and models. We now demonstrate this by showing how to define 'indistinguishability game'-based definitions, i.e., confidentiality-related specifications.

4.3.1 Defining Confidentiality-Related Operations

To support confidentiality, we define the set $\mathcal{X}$ to include the following three operations: 'Flip', 'Challenge', 'Guess'.

- 'Flip': selects a uniformly random bit $s_{\mathcal{X}}.b$ via coin flip, i.e., $s_{\mathcal{X}}.b \xleftarrow{\mathbf{R}} \{0,1\}$.
- 'Challenge': executes a desired operation with *one out of two possible inputs*, according to the value of $s_{\mathcal{X}}.b$. Namely, when $\mathcal{A}$ outputs $opr[e]$ = 'Challenge', the execution process invokes:

$$\mathcal{P}[inp[e].opr]\left(s_{ent[e]}, inp[e].inp[s_{\mathcal{X}}.b], clk[e]\right)$$

  where $inp[e].opr \in \mathcal{P}$ (one of the operations in $\mathcal{P}$) and $inp[e].inp$ is an 'array' with two possible inputs, of which only one is randomly chosen via $s_{\mathcal{X}}.b$, hence, the $inp[e].inp[s_{\mathcal{X}}.b]$ notation.
- 'Guess': checks if a 'guess bit', which is provided by the adversary as input, is equal to $s_{\mathcal{X}}.b$, and returns the result in *sec-out*$[e]$. Notice that the result must be in *sec-out*, so the adversary would not be able to access it.

These three operations are used as follows. The 'Flip' operation provides **Exec** with access to a random bit $s_{\mathcal{X}}.b$ that is not controlled or visible to $\mathcal{A}$.

Once the 'Flip' operation is invoked, the adversary can choose the 'Challenge' operation, i.e., $type[e] = \mathcal{X}$ and $opr[e] =$ 'Challenge', and $\mathcal{A}$ can specify any operation of $\mathcal{P}$ it wants to invoke ($inp[e].opr$) and any two inputs it desires ($inp[e].inp$). However, **Exec** will invoke $\mathcal{P}[inp[e].opr]$ with only one of the inputs, according to the value of the random bit $s_{\mathcal{X}}.b$, i.e., $inp[e].inp[s_{\mathcal{X}}.b]$; again, since $\mathcal{A}$ has no access to $s_{\mathcal{X}}.b$, $\mathcal{A}$ has no knowledge about which input is selected nor $\mathcal{A}$ can influence this selection. (As usual, further assumptions about the inputs can be specified using a model.) Then, $\mathcal{A}$ can choose the 'Guess' operation and provide its guess of the value of $s_{\mathcal{X}}.b$ (0 or 1) as input.

4.3.2 $\pi_{\mathsf{MsgConf}}{}^{IND}$: Message Confidentiality for Encrypted Communication

To illustrate how the aforementioned operations can be used in practice, we define the indistinguishability requirement predicate π^{IND} in Algorithm 5. π^{IND} checks that the adversary invoked the 'Guess' operation during the last event of the execution and examines whether the 'Guess' operation outputted $\top$ in its secure output and whether the π model was satisfied. The adversary 'wins' against this predicate when it guesses correctly during the 'Guess' event. Since an output of $\bot$ by a predicate corresponds to the adversary 'winning' (see, e.g., Def. 1), the π^{IND} predicate returns the *negation* of whether the adversary guessed correctly during the last event of the execution.

Algorithm 5 $\pi^{IND}(T)$ Predicate

```
1: return ¬(
2:     T.type[T.e] = 'X'
3:     and T.opr[T.e] = 'Guess' and T.sec-out[T.e] = ⊤      ▷ The last event is a 'Guess' event and A guessed correctly
4:     and π(T)                                             ▷ The model predicate π was met
       )
```

We can use π^{IND} to define more specific predicates; for example, we can use the π_{MsgConf} predicate, shown below, to define $\pi_{\mathsf{MsgConf}}{}^{IND}$, which can be used to define message confidentiality for an encrypted communication protocol. Namely, assume $\mathcal{P}$ is an encrypted communication protocol, which includes the following two operations: (1) a 'Send' operation which takes as input a message m and entity i_R and outputs an encryption of m for i_R, and a (2) 'Receive' operation, which takes as input an encrypted message and decrypts it.

The π_{MsgConf} model predicate (Algorithm 6) ensures that:

- $\mathcal{A}$ only asks for 'Send' challenges (since we are only concerned with whether or not $\mathcal{A}$ can distinguish outputs of 'Send').
- During all 'Send' challenges, messages are only sent from one specific entity i_{S} to one specific entity i_{R}.

– During each 'Send' challenge, $\mathcal{A}$ specifies two messages of equal length and the same recipient in the two possible inputs. This ensures that $\mathcal{A}$ does not distinguish the messages based on their lengths.
– $\mathcal{A}$ does not use the 'Receive' operation to decrypt any output of a 'Send' challenge.

Algorithm 6 $\pi_{\mathsf{MsgConf}}(T)$ Predicate

```
return (
    ∀ê ∈ {1, ..., T.e} s.t.
        T.type[ê] = 'X' and T.opr[ê] = 'Challenge':                ▷ Every 'Challenge' event
            T.inp[ê].opr = 'Send'                                  ▷ was for 'Send' operations only
            and |T.inp[ê].inp[0].m| = |T.inp[ê].inp[1].m|          ▷ with equal length messages
            and ∃ i_S, i_R ∈ T.N s.t.                              ▷ There is one specific sender i_S
                                                                     and one specific receiver i_R
                T.inp[ê].inp[0].i_R = T.inp[ê].inp[1].i_R = i_R    ▷ i_R is the recipient for both mes-
                                                                     sages
                and T.ent[ê] = i_S                                 ▷ And i_S is the sender
                and ∄ ê' s.t.                                      ▷ And there is no event ê'
                    T.opr[ê'] = 'Receive'
                    and T.inp[ê'].c = T.out[ê].c                   Where A uses the 'Receive'
                    and T.ent[ê'] = i_R                            ▷ event to decrypt the output of
                    and T.inp[ê'].i_S = i_S                          the challenge
)
```

5 The Modularity Lemmas

In this section we present the *model and requirement modularity* lemmas as well as the *model and requirement monotonicity* lemmas. These lemmas concretely illustrate and formally prove our framework's intuitive modularity properties that facilitate the reuse of models, requirements and protocols.

In the following lemmas, we say that a model $\widehat{\mathcal{M}}$ is *stronger* than a model $\mathcal{M}$ (and $\mathcal{M}$ is *weaker* than $\widehat{\mathcal{M}}$) if the predicate of $\widehat{\mathcal{M}}$ includes the predicate of $\mathcal{M}$; that is, the predicate of $\widehat{\mathcal{M}}$ is a conjunction of one or more predicates, and one of these predicates is the predicate of $\mathcal{M}$. Similarly, we say that a requirement $\widehat{\mathcal{R}}$ is *stronger* than a requirement $\mathcal{R}$ (and $\mathcal{R}$ is *weaker* than $\widehat{\mathcal{R}}$) if the predicate of $\widehat{\mathcal{R}}$ includes the predicate of $\mathcal{R}$; that is, the predicate of $\widehat{\mathcal{R}}$ is a conjunction of one or more predicates, and one of these predicates is the predicate of $\mathcal{R}$. Basically, stronger models enforce more constraints on the adversary or other assumptions, compared to weaker ones, while stronger requirements represent more properties achieved by a protocol or scheme, compared to weaker ones.

5.1 Model Modularity

The model modularity lemmas give the relationships between stronger and weaker models. They allow us to shrink stronger models (assumptions) into weaker ones and to expand weaker models (assumptions) into stronger ones as needed - and as intuitively expected to be possible.

The first model modularity lemma is very straightforward and shows that if an adversary $\mathcal{A}$ satisfies a stronger model $\widehat{\mathcal{M}}$, then $\mathcal{A}$ satisfies models that are weaker than $\widehat{\mathcal{M}}$ and have the same base function (using the same operations set $\mathcal{X}$). Intuitively, this holds because in order to satisfy a conjunction of multiple predicates, each of the predicates must be individually satisfied.

Lemma 1 (Weaker model satisfaction).
For any set $\mathcal{X}$ of execution-process operations, any (weaker) model $\mathcal{M} = (\pi, \beta)$, and any predicate π', if an adversary $\mathcal{A}$ satisfies the (stronger) model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta)$ (with negligible advantage) then $\mathcal{A}$ satisfies $\mathcal{M}$ (with negligible advantage), namely:

$$\mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \tag{4}$$

Proof. See Appx. B.3.

The second model modularity lemma shows that if an adversary satisfies two weaker models, then $\mathcal{A}$ satisfies the stronger model that is obtained by taking the conjunction of the predicates and the addition of the base functions of the weaker models (using the same operations set $\mathcal{X}$). Intuitively, this makes sense, because if an adversary *fails* some predicate, say π, for *at most* β fraction of the time (or negligibly close), and also *fails* π' for *at most* β' fraction of the time, then the most it can fail *either one* is $\beta + \beta'$ fraction of the time (or negligibly close).

Lemma 2 (Stronger model satisfaction).
For any set $\mathcal{X}$ of execution-process operations and any two (weaker) models $\mathcal{M} = (\pi, \beta), \mathcal{M}' = (\pi', \beta')$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ (with negligible advantage), then $\mathcal{A}$ satisfies the 'combined' (stronger) model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta + \beta')$ (with negligible advantage), namely:

$$\left(\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \tag{5}$$

Proof. See Appx. B.3.

The third model modularity lemma shows that if a protocol satisfies a requirement under a weaker model, then it satisfies the same requirement under a stronger model with the same base function (using the same operations set $\mathcal{X}$). Of course, this is true, because if we are assuming everything that is included in the stronger model, then we are automatically assuming everything in the weaker model with the same base function (by Lemma 1), which implies that the protocol satisfies the requirement for such adversaries.

Lemma 3 (Requirement satisfaction under stronger model).

If a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ (with negligible advantage) under only $\mathcal{M} = (\pi, \beta)$ using the execution-process operations set $\mathcal{X}$, then for any predicate π', $\mathcal{P}$ ensures $\mathcal{R}$ (with negligible advantage) under the 'combined' model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta)$, using $\mathcal{X}$, namely:

$$\mathcal{P} \models_{poly}^{\mathcal{M}, \mathcal{X}} \mathcal{R} \Rightarrow \mathcal{P} \models_{poly}^{\widehat{\mathcal{M}}, \mathcal{X}} \mathcal{R} \tag{6}$$

Proof. See Appx. B.3.

5.2 Model Monotonicity

We often may want to use the model modularity lemma to combine two models, $\mathcal{M}$ and $\mathcal{M}'$, but cannot, since they have different base functions $\beta \neq \beta'$. In such a case, we can first use the following model monotonicity lemma to have the same base function and then may apply the modularity lemma. Unfortunately, this requires using the base function $\hat{\beta}$, which is defined as the greater of β, β' for any given input $params$, i.e.:

$$(\forall params)\ \hat{\beta}(params) \equiv \max\{\beta(params), \beta'(params)\} \tag{7}$$

Lemma 4 (Model monotonicity). *Let $\mathcal{X}$ be any set of execution-process operations, π be a predicate, and β, β' be two base functions. Define the function $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all params. Let $\mathcal{M} = (\pi, \beta)$ and $\widehat{\mathcal{M}} = (\pi, \hat{\beta})$. Then holds:*

$$(\forall \mathcal{A}) \mathcal{A} \models_{poly}^{\mathcal{X}} \mathcal{M} \Rightarrow \mathcal{A} \models_{poly}^{\mathcal{X}} \widehat{\mathcal{M}} \tag{8}$$

Proof. See Appx. B.3.

5.3 Requirement Modularity

The requirement modularity lemmas give relationships between stronger and weaker requirements, assuming the same model $\mathcal{M}$ and operations set $\mathcal{X}$. They allow us to infer that a protocol satisfies a particular weaker requirement given that it satisfies a stronger one, or that a protocol satisfies a particular stronger requirement given that it satisfies weaker ones.

The first requirement modularity lemma shows that if a protocol satisfies a stronger requirement, then it satisfies a weaker one with the same base function (under the same model $\mathcal{M}$ and using the same operations set $\mathcal{X}$). That is, if protocol $\mathcal{P}$ satisfies requirement $\widehat{\mathcal{R}} = (\pi \wedge \pi', \beta)$, then $\mathcal{P}$ satisfies requirement $\mathcal{R} = (\pi, \beta)$. Similarly to Lemma 1, this holds because satisfying a conjunction of predicates implies satisfying each of the sub-predicates.

Lemma 5 (Weaker requirement satisfaction).

For any set $\mathcal{X}$ of execution-process operations, any model $\mathcal{M}$, any (weaker) requirement $\mathcal{R} = (\pi, \beta)$, and any predicate π', if a protocol $\mathcal{P}$ ensures the (stronger) requirement $\widehat{\mathcal{R}} \equiv (\pi \wedge \pi', \beta)$ (with negligible advantage) under model $\mathcal{M}$, then $\mathcal{P}$ ensures $\mathcal{R}$ (with negligible advantage) under model $\mathcal{M}$, namely:

$$\mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \mathcal{R} \tag{9}$$

Proof. See Appx. B.3.

The second requirement modularity lemma shows that if a protocol satisfies two weaker requirements, then it satisfies the stronger requirement that is obtained by taking the conjunction of the predicates and the addition of the base functions of the weaker requirements (under the same model $\mathcal{M}$ and using the same operations set $\mathcal{X}$). That is, if protocol $\mathcal{P}$ satisfies requirement $\mathcal{R} = (\pi, \beta)$ and $\mathcal{P}$ satisfies requirement $\mathcal{R}' = (\pi', \beta')$, then $\mathcal{P}$ satisfies requirement $\widehat{\mathcal{R}} = (\pi \wedge \pi', \beta + \beta')$. Similarly to Lemma 2, the intuition is that if an protocol fails some predicate, say π, for at most β fraction of the time (or negligibly close), and also fails π' for at most β' fraction of the time, then the most it can fail either one is $\beta + \beta'$ fraction of the time (or negligibly close).

Lemma 6 (Stronger requirement satisfaction).

For any set $\mathcal{X}$ of execution-process operations, any model $\mathcal{M}$, and any two (weaker) requirements $\mathcal{R} = (\pi, \beta), \mathcal{R}' = (\pi', \beta')$, if a protocol $\mathcal{P}$ ensures both $\mathcal{R}$ and $\mathcal{R}'$ (with negligible advantage) under model $\mathcal{M}$, then $\mathcal{P}$ ensures the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv (\pi \wedge \pi', \beta + \beta')$ (with negligible advantage) under model $\mathcal{M}$, namely:

$$\left(\mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \mathcal{R} \wedge \mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \widehat{\mathcal{R}} \tag{10}$$

Proof. See Appx. B.3.

5.4 Requirement Monotonicity

We may want to use the requirement modularity lemma to combine two requirements, $\mathcal{R}$ and $\mathcal{R}'$, but cannot, since they have different base functions $\beta \neq \beta'$. In such a case, we can first use the following requirement monotonicity lemma to have the same base function and then may apply the modularity lemma. Unfortunately, this requires using the base function $\hat{\beta}$, which is defined as the greater of β, β' for any given input *params*, as given in Equation 7.

Lemma 7 (Requirement monotonicity). *Let $\mathcal{X}$ be any set of execution-process operations, $\mathcal{M}$ be a model, π be a predicate, and β, β' be two base functions. Define the function $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all params. Let $\mathcal{R} = (\pi, \beta)$ and $\widehat{\mathcal{R}} = (\pi, \hat{\beta})$. Then holds:*

$$\mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \mathcal{R} \Rightarrow \mathcal{P} \models_{poly}^{\mathcal{M},\mathcal{X}} \widehat{\mathcal{R}} \tag{11}$$

Proof. See Appx. B.3.

6 Ensuring Polynomial-Time Runtime

In most of this work, as in most works in cryptography, we focus on PPT algorithms. For instance, consider Definition 2, where we require $\mathcal{A}, \mathcal{X}, \mathcal{P} \in PPT$, and the definition uses the concept of negligible advantage to refer to advantage functions which are smaller than any positive polynomial in the length of the inputs. However, when analyzing interacting systems as facilitated by MoSS, there is a concern: each of the algorithms might be in PPT, yet the runtime can be engineered to be *exponential in the number of steps*. For example, consider an adversary $\mathcal{A}$, that whenever it executes at the end of the execution-process loop (line 8 of Algorithm 1), its output state $s_{\mathcal{A}}$ is twice as long as it was in the input. Namely, if the size of the adversary's state in the beginning was $|s_{\mathcal{A}}|$, then after e rounds of the execution-process loop, the length of the outputted state would be $2^e \cdot |s_{\mathcal{A}}|$. Therefore, as $s_{\mathcal{A}}$ is provided as input to $\mathcal{A}$ and e is determined by $\mathcal{A}$, this shows that its runtime may be exponential in the original inputs to $\mathcal{A}$.

To ensure polynomial runtime, we provide the following. First, in Section 6.1, we present the $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}$ execution process, which is an extension of the $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process that allows specifications to refer to the runtime of algorithms invoked during the execution. Then, in Section 6.2, we define the $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ and $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}}$ model specifications, that enforce polynomial time restriction on adversary $\mathcal{A}$. In Section 6.3, we define the $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ requirement specification, that require a polynomial runtime from protocol $\mathcal{P}$. Finally, in Section 6.4 we show that if $\mathcal{A}$ and $\mathcal{P}$ satisfy these specifications, the total runtime in $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}$ is ensured to be polynomial.

6.1 The $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}$ execution Process

The $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}$ execution process (Algorithm 7) is identical to the $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process discussed in Section 2.2, but it has two additional variables: $sLog[\cdot]$ and $maxLen[\cdot]$. The $sLog[\cdot]$ variable includes $sLog[\cdot][1]$, which stores the final size of $s_{\mathcal{A}}$ in each iteration, and $sLog[\cdot][2]$, which stores the final size of $s_{ent[\cdot]}$ in each iteration. The $maxLen[\cdot]$ variable stores the maximum size of input given to $\mathcal{P}$; this makes it easy to compare the sizes of the outputs of $\mathcal{P}$ to the maximum input size. The only changes are the addition of line 3 to store the initial sizes of the states in $sLog[0]$, line 9 to save the $maxLen[e]$ value for each event e, line 16 to store the sizes of the output states, and the addition of $sLog[\cdot]$ and $maxLen[\cdot]$ to the execution transcript T in line 18.

6.2 The $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ and $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}}$ Models

We define the $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ model, defined as $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}} = (\pi^{\mathcal{A}}_{\text{polyGrow}}, 0)$, where the $\pi^{\mathcal{A}}_{\text{polyGrow}}$ predicate (Algorithm 8) restricts the size of each output of $\mathcal{A}$ in line 8 to be at most $T.sLog[0][1]$ greater than the size of the input to $\mathcal{A}$, and restricts the size of the new state $s_{\mathcal{A}}$ output by $\mathcal{A}$ in line 15 to be at most

Algorithm 7 Adversary-Driven Execution Process $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$

1: $(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}[\text{'Init'}](params.\mathcal{A}, params.\mathcal{P})$ ▷ *Initialize adversary with* $params.\mathcal{A}, params.\mathcal{P}$

2: $\forall i \in \mathsf{N}: \; s_i \leftarrow \mathcal{P}[\text{'Init'}]\,(\bot, (i, params.\mathcal{P}), \bot)$ ▷ *Initialize entities' local state*

3: $\boldsymbol{sLog[0] \leftarrow (|s_{\mathcal{A}}|, \max_{i \in \mathsf{N}}\{|s_i|\})}$

4: $s_{\mathcal{X}} \leftarrow params$ ▷ *Initial exec state*

5: $e \leftarrow 0$ ▷ *Initialize loop's counter*

6: **repeat**

7: $\quad e \leftarrow e + 1$ ▷ *Advance the loop counter*

8: $\quad (ent[e], opr[e], type[e], inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$ ▷ *$\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event e*

9: $\quad \boldsymbol{maxLen[e] \leftarrow \max\{|opr[e]|, |inp[e]|, |clk[e]|, |s_{ent[e]}|\}}$

10: $\quad$ **if** $type[e] =$ '$\mathcal{X}$' **then** ▷ *If $\mathcal{A}$ chose to invoke an operation from $\mathcal{X}$.*

11: $\quad\quad (s_{\mathcal{X}}, s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{X}\,[opr[e]]\,(s_{\mathcal{X}}, s_{ent[e]}, inp[e], clk[e], ent[e])$

12: $\quad$ **else** ▷ *$\mathcal{A}$ chose to invoke an operation from $\mathcal{P}$.*

13: $\quad\quad (s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$

14: $\quad$ **end if**

15: $\quad (s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}\,(s_{\mathcal{A}}, out[e])$ ▷ *$\mathcal{A}$ decides when to terminate the loop ($out_{\mathcal{A}} \neq \bot$), based on $out[e]$*

16: $\quad \boldsymbol{sLog[e] \leftarrow (|s_{\mathcal{A}}|, |s_{ent[e]}|)}$ ▷ **Save lengths of $s_{\mathcal{A}}$ and $s_{ent[e]}$ to return as part of T**

17: **until** $out_{\mathcal{A}} \neq \bot$

18: $T \leftarrow \left(\begin{smallmatrix} out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], type[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], sec\text{-}out[\cdot][\cdot] \\ , sLog[\cdot], maxLen[\cdot] \end{smallmatrix}\right)$

19: Return T ▷ *Output transcript of run*

$T.sLog[0][1]$ more than the size of the largest input to $\mathcal{A}$, which is either the initial $s_{\mathcal{A}}$ or $out[\hat{e}]$, for every event $\hat{e}$ of the execution. Note that $T.sLog[0][1]$ is the length of the initial state $s_{\mathcal{A}}$ output by the adversary in line 1 of Algorithm 7, so $\mathcal{A}$ itself defines this value at the beginning of the execution, but the value remains fixed for that particular execution.

Algorithm 8 The $\pi^{\mathcal{A}}_{\text{polyGrow}}$ (T) Predicate

1: **return** (

2: $\quad \forall \hat{e} \in \{1, \ldots, T.e\}$: ▷ *For each event*

3: $\quad\quad \forall y \in \{T.ent[\hat{e}], T.opr[\hat{e}], T.type[\hat{e}], T.inp[\hat{e}], T.clk[\hat{e}], T.\tau[\hat{e}]\}$: ▷ *And for each output of $\mathcal{A}$ from its first invocation during that event*

4: $\quad\quad\quad |y| \leq T.sLog[\hat{e}-1][1] + T.sLog[0][1]$ ▷ *The size is at most $T.sLog[0][1]$ greater than the size of $s_{\mathcal{A}}$ input to $\mathcal{A}$*

5: $\quad\quad$ **and** $T.sLog[\hat{e}][1] \leq \max\{T.sLog[\hat{e}-1][1], |T.out[\hat{e}]|\} + T.sLog[0][1]$ ▷ *The size of the output $s_{\mathcal{A}}$ is at most $T.sLog[0][1]$ greater than the size of the largest input to $\mathcal{A}$ in its second invocation*

)

We also define the $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}}$ model as $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}} = (\pi^{\mathcal{A}}_{\text{polySteps}}, 0)$, where $\pi^{\mathcal{A}}_{\text{polySteps}}$ simply checks that the number of events in the execution does not exceed $T.sLog[0][1]$. As mentioned earlier, $T.sLog[0][1]$ is the length of the initial state $s_{\mathcal{A}}$ output by the adversary in line 1 of Algorithm 7.

Algorithm 9 The $\pi^{\mathcal{A}}_{\text{polySteps}}$ (T) Predicate

1: **return** $T.e \leq T.sLog[0][1]$ ▷ *Number of steps is at most $T.sLog[0][1]$*

6.3 The $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ Requirement

We now define the $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ requirement as $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}} = (\pi^{\mathcal{P}}_{\text{polyGrow}}, 0)$, where $\pi^{\mathcal{P}}_{\text{polyGrow}}$, shown in Algorithm 10, restricts the size of each of the outputs of $\mathcal{P}$ to be, at most, $T.sLog[0][2]$ more than the size of the largest input to $\mathcal{P}$ (which is saved in the execution transcript in $T.maxLen[\cdot]$), for every event $\hat{e}$ of the execution. Note that $T.sLog[0][2]$ is the length of the largest state s_i output by the protocol in line 2 of Algorithm 7, so $\mathcal{P}$ defines this value at the beginning of the execution, but the value remains fixed for that particular execution.

Algorithm 10 The $\pi^{\mathcal{P}}_{\text{polyGrow}}$ (T) Predicate

1: **return** (

2: $\quad \forall \hat{e} \in \{1, \ldots, T.e\}$: ▷ *For each event*

3: $\quad\quad \forall y \in \{T.sLog[\hat{e}].s_{ent}, T.out[\hat{e}], T.sec\text{-}out[\hat{e}][\cdot]\}$: ▷ *The size of each output of $\mathcal{P}$*

4: $\quad\quad\quad |y| \leq T.maxLen[\hat{e}] + T.sLog[0][2]$ ▷ *Is at most $T.sLog[0][2]$ greater than the size of the largest input to $\mathcal{P}$*

)

6.4 Polynomial-growth PPT Algorithms and Properties

We now show that if $\mathcal{A}$ satisfies $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ and $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}}$ and $\mathcal{P}$ satisfies $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$, then the total running times of $\mathcal{A}$ and $\mathcal{P}$ will be polynomial - i.e., $\mathcal{A}$ cannot get an overall exponential execution time.

We begin with the definition of a stateful execution of an algorithm.

Definition 4 (Stateful execution of an algorithm $\mathcal{Z}$). *Let $\mathcal{Z}$ be an algorithm. A stateful execution of $\mathcal{Z}$ is a sequence of $N = \{1, \ldots, n\}$ invocations of $\mathcal{Z}$ such that for each $i \in \{2, \ldots, n\}$, the input to $\mathcal{Z}$ in the i^{th} invocation includes the* state *outputted by $\mathcal{Z}$ in the $(i-1)^{th}$ invocation, where the state is some well-defined part of each output of $\mathcal{Z}$.*

Next, we define a stateful execution in which there is only one initialization, at the beginning.

Definition 5 (Singly-initialized stateful execution of $\mathcal{Z}$ with *params*). *We define a* singly-initialized stateful execution of $\mathcal{Z}$ with *params as a stateful execution of $\mathcal{Z}$ for $N = \{1, \ldots, n\}$ invocations, where the first invocation initializes $\mathcal{Z}$ with params and for each $i \in \{2, \ldots, n\}$, the i^{th} invocation does not re-initialize $\mathcal{Z}$. We denote the set of all such possible executions as $E^{sis}_{\mathcal{Z},params}$.*

Next, we define a stateful execution with a single initialization operation where all invocations' outputs are bounded by the size of the first invocation.

Definition 6 (ℓ-bounded singly-initialized stateful execution of $\mathcal{Z}$ with *params*). *Let $\mathcal{E} \in E^{sis}_{\mathcal{Z},params}$ be a singly-initialized stateful execution of $\mathcal{Z}$ with params for $N = \{1, \ldots, n\}$ invocations, and let ℓ' be the length of the state outputted by $\mathcal{Z}$ in its first invocation during $\mathcal{E}$. We say that $\mathcal{E}$ is an* ℓ-bounded singly-initialized stateful execution of $\mathcal{Z}$ with *params, if $\ell' = \ell$ and for each $i \in \{2, \ldots, n\}$, the size of each of the outputs in the i^{th} invocation is at most ℓ greater than the maximum size of any of the inputs.*

Lastly, we define a *polynomial-growth PPT (PgPPT)* algorithm in Definition 7. Intuitively, PgPPT algorithms are algorithms which commit to a 'growth value' at the beginning of an execution and then, in each subsequent invocation, only have outputs at most as large as the largest input plus this growth value.

Definition 7 (Polynomial-growth PPT (PgPPT) algorithm). *Let $\mathcal{Z}$ be a PPT algorithm. We say that $\mathcal{Z}$ is a* polynomial-growth PPT (PgPPT) algorithm *if there exists a polynomial $p(\cdot)$ such that for every parameter params, for every $\mathcal{E} \in E^{sis}_{\mathcal{Z},params}$, $\mathcal{E}$ is a $p(params)$-bounded singly-initialized stateful execution.*

We can show that if $\mathcal{A}$ is a PgPPT algorithm, then it satisfies the $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ model, and if $\mathcal{P}$ is a PgPPT algorithm, then it satisfies the $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ requirement. See Lemmas 8 and 9.

Lemma 8. *Let $\mathcal{A}$ be a polynomial-growth PPT algorithm, and let $\mathcal{X}$ be any set of execution operations. Then $\mathcal{A}$ satisfies model $\mathcal{M}^{\mathcal{A}}_{polyGrow}$ with negligible advantage using execution operations $\mathcal{X}$.*

Lemma 9. *Let $\mathcal{P}$ be a polynomial-growth PPT algorithm, let $\mathcal{X}$ be any set of execution operations, and let $\mathcal{M}$ be any model. Then $\mathcal{P}$ satisfies requirement $\mathcal{R}^{\mathcal{P}}_{polyGrow}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$.*

Significantly, as stated in Lemma 10, we can show that if $\mathcal{A}$ satisfies the $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$ model and the $\mathcal{M}^{\mathcal{A}}_{\text{polySteps}}$ model and $\mathcal{P}$ satisfies the $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ requirement, then the running times of $\mathcal{A}$ and $\mathcal{P}$ during executions of the MoSS execution process will be polynomial.

Lemma 10. *Let $\mathcal{X}$ be a polynomial-growth PPT algorithm, let $\mathcal{A}$ be a PPT algorithm such that $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}^{\mathcal{A}}_{polyGrow}$, and let $\mathcal{P}$ be a PPT algorithm such that $\mathcal{P} \models^{\mathcal{M}^{\mathcal{A}}_{polyGrow},\,\mathcal{X}}_{poly} \mathcal{R}^{\mathcal{P}}_{polyGrow}$. Then, during the execution of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$, the total running time of $\mathcal{A}$ and the total running time of $\mathcal{P}$ are both bounded by a polynomial in params.*

7 Concrete Security

Extending the MoSS framework to support concrete security definitions is relatively simple, as we show next. In concrete security, the adversary's advantage is bounded by a specific function in the bounds on the 'adversary resources', which may include different types of resources such as the runtime (in a specific computational model), length (of inputs, keys, etc.), and the number of different operations that the adversary invokes (e.g., 'oracle calls'). This approach is compatible with our definition of the advantage $\epsilon^{\xi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ of $\mathcal{A}$ against $\mathcal{P}$ for ξ and $\mathcal{X}$ (Definition 1), where the advantage is a function of the parameters *params*. However, some relatively minor challenges remain.

First, in the existing concrete definitions, the definition of 'adversary resources' is an input to the advantage function. We considered adding it to our definition of the advantage function by defining appropriate functions over the transcript T; however, the advantage bounds the *probability* of executions so it cannot use a measurement of the value in a particular execution. Instead, we chose to specify these bounds as *additional parameters* in *params*, denoted as $params.\mathcal{A}.bounds$, and provide these parameters to the adversary. Then, our model predicates can easily validate that the adversary does not *exceed* these bounds. Note that *params* is already an input to the advantage function, addressing the first challenge. In Sec. 7.2, we define an appropriate (simple) model that validates that the adversary does not exceed the bounds. For most bounds, e.g., on the number of different operations, this can be easily validated by evaluating the transcript T.

However, this brings us to the second challenge: bounding the runtime. Clearly, it is not possible to compute the adversary's runtime from the transcript T of the execution process (Algorithm 1). To allow concrete security which also considers runtime, we needed to extend that execution process. We now describe this extension, which is simpler than expected.

7.1 Using StepCount for Runtime-Measuring Execution Process

The advantage function of concrete security reductions may refer to a bound on the adversary's runtime (using some fixed computational model). To allow such bounds, we now define a simple modification to our use of the execution process. **Exec** would return, as a part of the transcript T, the actual runtime of the adversary $\mathcal{A}$. This allows model predicates to validate that the adversary did not exceed the bound on its runtime, provided in $params.\mathcal{A}$.

The solution uses a well-known technique from the theory of complexity: run $\mathcal{A}$ *'within'* another algorithm that returns the runtime of $\mathcal{A}$, together with the 'regular' outputs of $\mathcal{A}$. Specifically, let StepCount denote an algorithm that receives another algorithm, which we conveniently denote $\mathcal{A}$ (as we would in fact use the adversary $\mathcal{A}$ as input to StepCount). Namely, in the concrete definitions, we use $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$ instead of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$. It remains to describe the operation of StepCount.

Note that StepCount maintains its own state, which contains, as part of it, the state of the adversary $\mathcal{A}$. This creates a somewhat complex situation, which may be familiar to the reader from constructions in the theory of complexity (or, to the practitioner, from the relation between a virtual machine and the program it is running) - yet a bit confusing. Namely, the execution process received the algorithm $\mathsf{StepCount}(\mathcal{A})$ as the adversary, while $\mathsf{StepCount}(\mathcal{A})$ is running the 'real' adversary $\mathcal{A}$.

This is especially confusing regarding the state of the adversary; the state $s_{\mathcal{A}}$ maintained by the execution process is now the state of the StepCount function. This state consists of two parts (variables); one of them is the state of the original adversary $\mathcal{A}$. We denote this variable by $s_{\mathcal{A}}.s_{\mathcal{A}}$; this unwieldy notation is trying to express the fact that from the point of view of the 'real' adversary $\mathcal{A}$, this is its (entire) state, while it is only part of the state $s_{\mathcal{A}}$ of the $\mathsf{StepCount}(\mathcal{A})$ algorithm (run by the execution process).

The other variable in the state $s_{\mathcal{A}}$ of StepCount, is a counter $s_{\mathcal{A}}.\mathsf{StepCount}$. Notice that this variable is invisible to $\mathcal{A}$ (since it is not part of $s_{\mathcal{A}}.s_{\mathcal{A}}$). The function StepCount uses $s_{\mathcal{A}}.\mathsf{StepCount}$ to sum-up the total runtime of $\mathcal{A}$. Namely, whenever the execution process invokes $\mathsf{StepCount}(\mathcal{A})$, then StepCount 'runs' $\mathcal{A}$ on the provided inputs, measuring the time (number of steps) until $\mathcal{A}$ returns its response, and adding it to $s_{\mathcal{A}}.\mathsf{StepCount}$.

When $\mathcal{A}$ returns a response, StepCount first increments the counter $s_{\mathcal{A}}.\mathsf{StepCount}$ by the run-time of $\mathcal{A}$ in this specific invocation. Next, StepCount checks if $\mathcal{A}$ signaled termination of the execution process. When $\mathcal{A}$ signals termination (by returning $out_{\mathcal{A}} \neq \perp$), then StepCount sets $out_{\mathcal{A}}.\mathsf{StepCount} \leftarrow s_{\mathcal{A}}.\mathsf{StepCount}$, i.e., adds to $out_{\mathcal{A}}$ the computed total run-time of $\mathcal{A}$ during this execution[7]; of course, we still have $out_{\mathcal{A}} \neq \perp$ and therefore the execution process terminates - returning the total runtime of $\mathcal{A}$ as part of $out_{\mathcal{A}}$. Although the runtime is carried in $out_{\mathcal{A}}$, the adversary cannot modify it.

[7] Note this would override any value that $\mathcal{A}$ may write on $out_{\mathcal{A}}.\mathsf{StepCount}$, i.e., we essentially forbid the use of $out_{\mathcal{A}}.\mathsf{StepCount}$ by $\mathcal{A}$.

Note that this solution *does not requires any changes to the execution process* $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$; we only *changed the adversary*. For the same reason the use of StepCount does not change any other property of the execution - including the runtime of the adversary $\mathcal{A}$.

7.2 Enforcing Concrete Security via Models

We will now illustrate how model predicates can be defined to enforce concrete security bounds, following the adjustments discussed in Sec. 7.1. Suppose *params* includes a structure called $params.\mathcal{A}.bounds.maxCalls$, where $params.\mathcal{A}.bounds.maxCalls[type][opr]$ contains the maximum number of calls to an operation *opr* of type *type* that should not be exceeded by the adversary. Also suppose *params* includes a value called $params.\mathcal{A}.bounds.maxSteps$, which is the maximum number of steps that the adversary is allowed to take. Finally, suppose we are using StepCount, i.e., executing $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$. Then, we can use the π^{Bounds} model predicate (Algorithm 11) to ensure that: (1) $\mathcal{A}$ does not exceed the bounds on the number of calls to each operation that is part of $params.\mathcal{A}.bounds.maxCalls$, and (2) the bound on the number of steps taken by $\mathcal{A}$ is enforced.

Algorithm 11 $\pi^{\text{Bounds}}(T)$ Predicate

return (

$\forall\ type \in params.\mathcal{A}.bounds.maxCalls$:

  $\forall\ opr \in params.\mathcal{A}.bounds.maxCalls[type]$: ▷ *The number of calls to each operation with bounds is not exceeded*

$$\left|\left\{ \hat{e} \;\middle|\; \begin{array}{l} \hat{e} \in \{1,\ldots,T.e\} \textbf{ and} \\ T.type[\hat{e}] = type \textbf{ and} \\ T.opr[\hat{e}] = opr \end{array} \right\}\right| \leq params.\mathcal{A}.bounds.maxCalls[type][opr]$$

and $T.out_{\mathcal{A}}.\mathsf{StepCount} \leq params.\mathcal{A}.bounds.maxSteps$ ▷ *The number of steps taken by $\mathcal{A}$ is not exceeded*

)

7.3 Definitions

Finally, in this section, we adjust Definitions 1, 2 and 3 for concrete security analysis. To that end, we first describe the MaxAdvantage function ϵ.

A MaxAdvantage function ϵ is the maximum *advantage* probability that the adversary may have (for a given specification), *over* the 'base' probability. Namely, specification $\xi = (\pi, \beta)$ is *satisfied with MaxAdvantage* ϵ as long as for every *params* used in the definition, the adversary's success probability minus $\beta(params)$ is bounded by $\epsilon(params)$. The sum $\beta(params) + \epsilon(params)$ for any such *params* should be in the range $[0, 1]$.

Thus, in the concrete definition, a specification may be satisfied *with MaxAdvantage* ϵ, where the MaxAdvantage may be a function of any of the parameters

included in *params*. Note, however, that if we want to use a MaxAdvantage function that depends on the bounds restricting the adversary (i.e., the bounds in $params.\mathcal{A}.bounds$), then we need to assume a model that ensures that the adversary really does not exceed these bounds.

We now define the ξ-StepCount-*advantage* for a given specification $\xi = (\pi, \beta)$ and a given value of *params* as the probability *over* $\beta(params)$ that the predicate π, applied to a random resulting execution transcript $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$ and *params*, is not satisfied.

Definition 8 (Advantage of $\mathcal{A}$ against $\mathcal{P}$ for specification ξ and execution-operations set $\mathcal{X}$ using StepCount). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X} \in PPT$ and let $\xi = (\pi, \beta)$ be a specification. The advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification ξ and execution-operations set $\mathcal{X}$ using* StepCount *is defined as*

$$\epsilon^{\xi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \stackrel{def}{=} \Pr\left[\begin{array}{l}\pi\left(T\right) = \perp,\ where\\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)\end{array}\right] - \beta(params) \tag{12}$$

We now give the concrete definition of model-satisfying adversary; note that here we use a model $\mathcal{M}$ as the specification ξ of Definition 8.

Definition 9 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with MaxAdvantage ϵ using execution operations $\mathcal{X}$). *Let $\mathcal{A}$ be a PPT algorithm, and let $\mathcal{M} = (\pi, \beta)$ be a model specification. Let ϵ be a function $\epsilon(params) : \{0,1\}^* \to [0,1]$, such that, for any given params, holds $\beta(params) + \epsilon(params) \in [0,1]$. Let $\mathcal{X}$ be a set of operations provided by the execution process* **Exec**. *We say that adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with MaxAdvantage ϵ using execution operations $\mathcal{X}$, denoted as $\mathcal{A} \models^{\mathcal{X}}_{\epsilon} \mathcal{M}$, if for every protocol $\mathcal{P} \in PPT$ and params, the $\epsilon^{\mathcal{M},\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ advantage is bounded by $\epsilon(params)$.*

Similarly, we can define requirements with respect to a MaxAdvantage function. The following is the concrete definition of requirement-satisfying protocol.

Definition 10 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ with MaxAdvantage ϵ under model $\mathcal{M}$ with advantage $\epsilon_{\mathcal{M}}$ using execution operations $\mathcal{X}$). *Let $\mathcal{R} = (\pi, \beta)$ be a requirement specification. Let $\epsilon, \epsilon_{\mathcal{M}}$ be functions such that $\epsilon^* \in \{\epsilon, \epsilon_{\mathcal{M}}\}$ is a function $\epsilon^*(params) : \{0,1\}^* \to [0,1]$, and for any given params, holds $\beta(params) + \epsilon^*(params) \in [0,1]$. Let $\mathcal{X}$ be a set of operations provided by the execution process* **Exec**. *Let $\mathcal{M}$ be a model. We say that protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ with MaxAdvantage ϵ under model $\mathcal{M}$ with advantage $\epsilon_{\mathcal{M}}$ using execution operations $\mathcal{X}$, denoted $\mathcal{P} \models^{\mathcal{M},\epsilon_{\mathcal{M}},\mathcal{X}}_{\epsilon} \mathcal{R}$, if for every PPT adversary $\mathcal{A}$ that satisfies $\mathcal{M}$ with advantage $\epsilon_{\mathcal{M}}$ using execution operations $\mathcal{X}$ and for every $params \in \{0,1\}^*$, the $\epsilon^{\mathcal{R},\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ advantage is bounded by $\epsilon(params)$.*

8 AuthBroadcast: Authenticated Broadcast Protocol

In this section, we present and analyze the AuthBroadcast protocol, as an example of the use of MoSS. AuthBroadcast is a simple authenticated broadcasting

protocol; it is not a contribution by itself, merely an example of the MoSS framework in action - although, the approach can be extended to analyze security of 'real' secure-communication protocols. To see the MoSS framework applied to a more complex protocol, see [16].

The AuthBroadcast protocol enables a set of entities N to broadcast authenticated messages to each other, i.e., to validate that a received message was indeed sent by a member of N. The protocol uses a standard deterministic message authentication scheme MAC which takes as input a tag length, key, and message and outputs a tag. (See Appendix A for the formal definition of the authentication scheme along with its security definition.)

The implementation of the AuthBroadcast protocol is detailed in Sec. 8.3. Briefly, the protocol includes five operations:

- 'Init' (Algorithm 17): initializes the local state of the entity.
- 'Key-Gen' (Algorithm 18): generates the shared authentication key and sends it to the rest of the entities.
- 'Sec-in' (Algorithm 19): receives a shared-key from another entity.
- 'Broadcast' (Algorithm 20): broadcasts an authenticated message.
- 'Receive' (Algorithm 21): receives incoming broadcast message, verify its authenticity and output it (to the application).

We first define the protocol's model specifications in Sec. 8.2, the desired security requirements in Sec. 8.1, and the formal definition of the protocol in Sec. 8.3. We conclude this section with a formal security analysis of the AuthBroadcast protocol in Sec. 8.4.

8.1 Security Requirements

We define two security requirements for the AuthBroadcast protocol. The first, $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$, ensures freshness and authenticity of received messages, and the second, $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$, ensures bounded-delay broadcasting.

We define the $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ requirement as $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta} = (\pi_{\mathsf{AuthComRcv}_\Delta}, \beta_{\mathsf{AuthComRcv}_\Delta})$, where $\beta_{\mathsf{AuthComRcv}_\Delta}(params) = 2^{-params.\mathcal{P}.n}$ and where the $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ predicate is defined in Sec. 4.2. The base function is $2^{-params.\mathcal{P}.n}$, because the AuthBroadcast protocol is implemented using some message authentication scheme MAC, which outputs $params.\mathcal{P}.n$-bit tags (i.e., we allow the adversary to have $2^{-params.\mathcal{P}.n}$ probability to forge a tag, which would make $\pi_{\mathsf{AuthComRcv}_\Delta}$ evaluate to $\perp$). Of course one could change to negligible probability by using the $params.\mathcal{P}.n$ to be the same as the security parameter.

The $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}} = (\pi_{\mathsf{Broadcast}_{\Delta_{com}}}, \beta_{\mathsf{Broadcast}_{\Delta_{com}}})$ requirement requires that all broadcast messages are correctly received at every other entity (except the sender of the broadcast) within Δ_{com} real time (unless the execution terminates before that time, of course). In this case, $\beta_{\mathsf{Broadcast}_{\Delta_{com}}} = 0$, and the $\pi_{\mathsf{Broadcast}_{\Delta_{com}}}$ requirement predicate is shown in Algorithm 12.

Algorithm 12 $\pi_{\mathsf{Broadcast}_{\Delta_{com}}}(T)$ Predicate

```
1: return (
2:   ∀ê_B ∈ {1, ..., T.e − 1} s.t.
3:       (T.opr[ê_B] = 'Broadcast' and                          ▷ For every 'Broadcast' event
4:       T.τ[T.e] ≥ T.τ[ê_B] + Δ_com):                          ▷ That the execution did not terminate Δ_com real time after it
5:         ∃ê_R ∈ {ê_B+1, ..., T.e}, ∀i ∈ N s.t. i ≠ T.ent[ê_B]  ▷ There is a later event for each entity except the broadcasting entity
6:                 where T.τ[ê_B] + Δ_com ≥ T.τ[ê_R]              ▷ Within Δ_com real time
7:                 and T.out[ê_R] = ('Receive', T.inp[ê_B].m)     ▷ Where the broadcast message was received correctly
8:                 and T.ent[ê_R] = i                             ▷ By the relevant entity
   )
```

8.2 AuthBroadcast Model Specifications

In the following subsections we present the $\mathcal{M}_{\text{SecKeyShare}}$ and $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ models, which are later used in the analysis of AuthBroadcast. In Sec. 8.2.1 we describe the shared-key setup model $\mathcal{M}_{\text{SecKeyShare}}$, which suffices for AuthBroadcast to ensure the $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ requirement (i.e., authentication of messages). In Sec. 8.2.2 we present $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$, a model which suffices for AuthBroadcast to ensure the $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ requirement (i.e., guaranteed delivery of messages).

8.2.1 The $\mathcal{M}_{\text{SecKeyShare}}$ Secure Shared-Key Initialization Model

We first describe the secure shared-key initialization model $\mathcal{M}_{\text{SecKeyShare}}$. We define $\mathcal{M}_{\text{SecKeyShare}} = (\pi_{\text{SecKeyShare}}, 0)$. Notice that $\mathcal{M}_{\text{SecKeyShare}}$ is a *generic* model and may be reused for analysis of other shared-key protocols.

The $\pi_{\text{SecKeyShare}}$ model predicate, shown in Algorithm 13, has two objectives. First, it ensures that only one entity securely shared a key during the protocol's execution. The reasoning behind this objective is simplicity. Namely, by ensuring that only one entity generated and shared an authentication key, we eliminate more complex scenarios where multiple entities shared a key or cases of key replacements. Obviously, such scenarios can be easily supported, however, it would introduce extra complexity which is not needed to demonstrate the framework.

The second goal of this predicate is to ensure that before any 'Broadcast' or 'Receive' operation was invoked on any entity $i \in \mathsf{N}$, that entity indeed received the shared key. To that end, the $\pi_{\text{SecKeyShare}}$ model predicate verifies that there was a relevant 'Sec-in' operation of type '$\mathcal{X}$' invoked on entity i. As discussed in Sec. 2.3, the 'Sec-in' $\mathcal{X}$-operation invokes the 'Sec-in' operation of $\mathcal{P}$ with the secure output of some event. The $\pi_{\text{SecKeyShare}}$ model predicate ensures that this operation was invoked on entity i with the relevant secure output of the event where the shared-key was generated.

Finally, we must also make sure that the adversary does not cause an entity to receive 'fake' securely shared values (using the 'Sec-in' operation). To do this, in line 5 of Algorithm 13, we evaluate the predicate $\pi^{\text{Excl}}_{\mathcal{P}[\text{'Sec-in'}]}$, which is separately shown in Algorithm 14. This ensures that the adversary cannot invoke the 'Sec-in' operation of $\mathcal{P}$ directly; instead, only invocations of the 'Sec-in' $\mathcal{X}$-operation are allowed. Consequently, only values that were truly returned through *sec-out* can be received using 'Sec-in'.

Algorithm 13 $\pi_{\text{SecKeyShare}}(T)$ Predicate

1: **return** $\top$ if (
2: $\exists\ \hat{e} \in \{1, \ldots, T.e\}$ **s.t.** $T.opr[\hat{e}] =$ 'Key-Gen' **and**
$\forall \hat{e}' \in \{1, \ldots, T.e\}$ **s.t.** $\hat{e}' \neq \hat{e}$: $T.opr[\hat{e}'] \neq$ 'Key-Gen' ▷ *Only one key was shared*
3: **and if** $T.opr[\hat{e}'] \in \{$'Broadcast','Receive'$\}$ ▷ *If the authentication key is used*
then $\exists\ \hat{e}'' \in \{\hat{e}+1, \ldots, \hat{e}'-1\}$ ▷ *Then* prior *to using the key*
4: **s.t.** $T.type[\hat{e}''] =$ '$\mathcal{X}$'
and $T.opr[\hat{e}''] =$ 'Sec-in'
and $T.ent[\hat{e}''] = T.ent[\hat{e}']$
and $T.inp[\hat{e}''] = \hat{e}$
▷ *The key was securely delivered to the relevant entity, i.e., the 'Sec-in' operation from $\mathcal{X}$ was invoked on that entity, delivering the secure output of the 'Key-Gen' operation to the relevant entity*
5: **and** $\pi^{\text{Excl}}_{\mathcal{P}[\text{'Sec-in'}]}$ ▷ *And $\mathcal{A}$ did not use $\mathcal{P}$['Sec-in'] directly*
)

Algorithm 14 $\pi^{\text{Excl}}_{\mathcal{P}[\text{'Sec-in'}]}(T)$ Predicate

1: **return** $\top$ if ($\nexists\ \hat{e} \in \{1, \ldots, T.e\}$ **s.t.** $T.type[\hat{e}] =$ '$\mathcal{P}$' **and** $T.opr[\hat{e}] =$ 'Sec-in')

8.2.2 The $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ Model

We now describe the model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}} = (\pi^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}, 0)$, where the $\pi^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ predicate is a conjunction of $\pi^{\text{Drift}}_{\Delta_{clk}}$ (defined in Sec. 3.3), $\pi_{\text{SecKeyShare}}$ (defined in Sec. 8.2.1), and additionally $\pi^{\text{Broadcast}}_{\Delta_{com}}$ and $\pi^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}$, which are defined in this section. Later, in Sec. 8.4, we claim that the AuthBroadcast protocol satisfies $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ (described in Sec. 8.1) with negligible advantage under model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$.

The $\pi^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ model predicate is a conjunction of four sub-predicates, i.e.:

$$\pi^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}} = \pi_{\text{SecKeyShare}} \wedge \pi^{\text{Drift}}_{\Delta_{clk}} \wedge \pi^{\text{Broadcast}}_{\Delta_{com}} \wedge \pi^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}} \tag{13}$$

Where $f(\Delta_{com}, \Delta_{clk}) = \Delta_{com} + 2\Delta_{clk}$.

That is, we assume that (1) a key is securely shared once among all entities ($\pi_{\text{SecKeyShare}}$), (2) real time is monotonically increasing and local time at all

entities is always within Δ_{clk} drift from the real time ($\pi^{\text{Drift}}_{\Delta_{clk}}$), (3) there is reliable, bounded-delay broadcast communication ($\pi^{\text{Broadcast}}_{\Delta_{com}}$), and (4) $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}.\Delta$, which is needed for the protocol to ensure receipt of valid packets, as explained below ($\pi^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$). We next define the $\pi^{\text{Broadcast}}_{\Delta_{com}}$ and $\pi^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$ predicates.

The $\pi^{\text{Broadcast}}_{\Delta_{com}}$ model predicate verifies that every 'Broadcast' message is received as input by every other entity (except the broadcasting one) within Δ_{com} real time (assuming that the execution did not end yet before that time). See Algorithm 15.

Algorithm 15 $\pi^{\text{Broadcast}}_{\Delta_{com}}$ (T) Predicate

return (
  $\forall \hat{e}_{\text{B}} \in \{1, \ldots, T.e - 1\}$:
    if $(T.out[\hat{e}_{\text{B}}] = (\text{'Broadcast'}, m, timeSent, tag)$ ▷ *If the output includes a broadcast message (with timestamp and tag)*
      and $T.\tau[T.e] \geq T.\tau[\hat{e}_{\text{B}}] + \Delta_{com}$ ▷ *And execution did not terminate yet after Δ_{com} real time*
    then $\forall i \in \mathsf{N}$ **s.t.** $i \neq T.ent[\hat{e}_{\text{B}}]$: ▷ *Then for each entity except the broadcasting entity*
      $\exists \hat{e}_{\text{R}} \in \{\hat{e}_{\text{B}} + 1, \ldots, T.e\}$ ▷ *There is a later event*
        and $T.\tau[\hat{e}_{\text{B}}] + \Delta_{com} \geq T.\tau[\hat{e}_{\text{R}}]$ ▷ *Within Δ_{com} real time*
        and $T.ent[\hat{e}_{\text{R}}] = i$ ▷ *Where the entity is i*
        and $T.opr[\hat{e}_{\text{R}}] = \text{'Receive'}$ ▷ *And which is a receive event*
        and $T.inp[\hat{e}_{\text{R}}] = (m, timeSent, tag)$ ▷ *And where the input is the broadcast message, timestamp, and tag*
)

The $\pi^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$ model predicate, shown in Algorithm 16, checks that the parameter $params.\mathcal{P}.\Delta$ is at least as high as some given function f of Δ_{com} and Δ_{clk}. For the AuthBroadcast protocol, we need to use the function $f(\Delta_{com}, \Delta_{clk}) = \Delta_{com} + 2\Delta_{clk}$. Thus, for this f, the $\pi^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$ predicate simply checks that $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}.\Delta$. Intuitively, this is necessary for the 'Broadcast' protocol to provide guaranteed delivery, because otherwise, although message packets may arrive, their timestamps might not be within the $params.\mathcal{P}.\Delta$ interval of the time at the receiver. Then, they would be considered old by the protocol and would not be received successfully.

Algorithm 16 $\pi^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$ (T) Predicate

return $(f(\Delta_{com}, \Delta_{clk}) \leq params.\mathcal{P}.\Delta)$

8.3 The AuthBroadcast Protocol

The AuthBroadcast protocol is a PPT algorithm with the operations:

$$(\text{'Init'}, \text{'Key-Gen'}, \text{'Sec-in'}, \text{'Broadcast'}, \text{'Receive'})$$

described in Algorithms 17-21. The protocol uses the following state variables in entity i: $s_i.1^\kappa$ (key length), $s_i.n$ (length of tags), $s_i.\Delta$ (maximal allowed delay, for freshness), and $s_i.k$ (authentication key).

Algorithm 17 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Init'}](s, inp, clk)$

1: $(i, params.\mathcal{P}) \leftarrow inp$
2: **if** $s = \perp$ **then** ▷ *This is the first call to 'Init'*
3: $\quad 1^\kappa \leftarrow params.\mathcal{P}.1^\kappa$ ▷ *Initialize key length*
4: $\quad n \leftarrow params.\mathcal{P}.n$ ▷ *Initialize tag length*
5: $\quad \Delta \leftarrow params.\mathcal{P}.\Delta$ ▷ *Initialize freshness interval*
6: $\quad k \leftarrow \perp$ ▷ *Initialize authentication key*
7: $\quad$ **return** $(1^\kappa, n, \Delta, k)$
8: **end if**
9: **return** $((1^\kappa, n, \Delta, k), \perp, \perp)$

Algorithm 18 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Key-Gen'}](s, inp, clk)$

1: $s.k \xleftarrow{\mathbf{R}} \{0,1\}^{|s.1^\kappa|}$ ▷ *Choose a shared key uniformly at random*
2: **return** $(s, \perp, s.k)$ ▷ *Share the key by returning it in sec-out*

Algorithm 19 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Sec-in'}](s, inp, clk)$

1: **if** $inp \neq \perp$ **then** $s.k \leftarrow inp$ ▷ *Save the shared key*
2: **return** $(s, \perp, \perp)$

Algorithm 20 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Broadcast'}](s, inp, clk)$

1: $m \leftarrow inp$
2: **if** $(s.k \neq \perp)$ **then**
3: $\quad timeSent \leftarrow clk$
4: $\quad tag \leftarrow \mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent)$ ▷ *Compute the tag over message and local time*
5: $\quad out = (m, timeSent, tag)$ ▷ *Return 'Broadcast', the message, local time, and tag*
6: **end if**
7: **return** $(s, out, \perp)$

Algorithm 21 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$['Receive']$(s, inp, clk)$

1: $(m, timeSent, tag) \leftarrow inp$; $out = \perp$
2: **if** $(s.k \neq \perp$
3: **and** $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent) = tag$ ▷ *Check if the tag is valid*
4: **and** $clk - timeSent \leq s.\Delta)$: ▷ *Check freshness*
5: $out =$ ('Receive', m) ▷ *If all Ok, output m*
6: **end if**
7: **return** $(s, out, \perp)$

8.4 Security Analysis

The MoSS framework allows the analysis of the same protocol under different models, as we demonstrate here. Specifically, we present the analysis of AuthBroadcast in several steps, where in each step, we prove that AuthBroadcast satisfies additional requirements - assuming increasingly stronger models:

1. We first show that AuthBroadcast ensures *authentication* of received messages under $\mathcal{M}_{\text{SecKeyShare}}$, the simple model for shared-key initialization described in Sec. 8.2.1; note that $\mathcal{M}_{\text{SecKeyShare}}$ is a *generic* model - it may be reused for analysis of other shared-key protocols. Namely, we show that AuthBroadcast satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShare}}$.
2. We then show that AuthBroadcast also ensures *freshness* of received messages under a stronger model that also assumes a weak-level of clock synchronization (bounded clock drift). Namely, we show that for any freshness interval Δ, AuthBroadcast satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under model $\mathcal{M} = (\pi, 0)$, where π is the conjunction of $\pi^{\text{Drift}}_{\Delta_{clk}}$ (Algorithm 3) and $\pi_{\text{SecKeyShare}}$ (Algorithm 13).
3. Finally, we show that AuthBroadcast also ensures *guaranteed bounded-delay delivery* of broadcast messages under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ (see Sec. 8.2.2), an even stronger model, that also assumes a bounded delay of communication. Specifically, we show that AuthBroadcast satisfies $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ with negligible advantage under model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$.

Note that by Lemma 3 (Sec. 5), it automatically follows that AuthBroadcast also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under the stronger models in points 2 and 3, and similarly, that AuthBroadcast satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under the stronger model in point 3. This is because all three of the models used in this analysis have the same base function and the predicates are built up incrementally to correspond to increasingly stronger assumptions about the adversary, synchronization, and communication channel. This shows one of the nice characteristics of the MoSS framework - having proven the above three properties, it is easy to show that, e.g., AuthBroadcast also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$.

We present each of the above three points below in Theorem 1, consisting of three claims, and we give proofs of the three claims in Sec. 8.4.1-8.4.3.

Theorem 1. *AuthBroadcast satisfies the properties given in Claims 1-3. Informally, this means that AuthBroadcast ensures authentication of received messages assuming shared-key initialization, ensures freshness and authentication assuming additionally bounded clock drift, and ensures bounded-delay delivery of broadcast messages assuming additionally a guaranteed, bounded-delay communication channel.*

Claim 1. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}_{\text{SecKeyShare}},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_{\infty}}$$

Where $\mathcal{M}_{\text{SecKeyShare}}$ is defined in Sec. 8.2.1 and $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ is described in Sec. 8.1.

Claim 2. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined in Sec. 2.3). Let $\Delta = params.\mathcal{P}.\Delta + 2\Delta_{clk}$. Let $\mathcal{M} = (\pi, 0)$, where $\pi = \pi_{\text{SecKeyShare}} \wedge \pi^{\text{Drift}}_{\Delta_{clk}}$. Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$$

Where $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ is described in Sec. 8.1.

Claim 3. Let MAC be a message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined in Sec. 2.3). Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$$

Where $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ is defined in Sec. 8.2.2 and $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ is defined in Sec. 8.1.

8.4.1 Proof of Claim 1

In this section, we prove Claim 1 of Theorem 1 - that AuthBroadcast ensures authentication of received messages assuming shared-key initialization. The claim is restated below.

Claim 1. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}_{\text{SecKeyShare}},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_{\infty}}$$

Where $\mathcal{M}_{\text{SecKeyShare}}$ is defined in Sec. 8.2.1 and $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ is described in Sec. 8.1.

We prove Claim 1 by contradiction - namely, by showing that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_{\infty}}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShare}}$ using execution operations $\mathcal{X}$, then MAC is not asymptotically UF.

Sketch of the Proof of Claim 1.

We complete the proof in three steps:

1. We first define an algorithm $\mathcal{A}'$ and describe how it works. This includes defining a modified version of $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, called $\overline{\mathsf{AuthBroadcast}}$, which uses $\mathsf{OTag}(\cdot)$ and and $\mathsf{OVer}(\cdot,\cdot)$ instead of $\mathsf{MAC}_{s.n}(s.k,\cdot)$ to tag and authenticate messages.
2. Then we show that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShare}}$ using execution operations $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}_{\text{SecKeyShare}}$ that for some value of $params$ has non-negligible advantage (for requirement $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ and execution-operations set $\mathcal{X}$) to 'win' against the $\overline{\mathsf{AuthBroadcast}}$ protocol (i.e., the $\overline{\mathsf{AuthBroadcast}}$ protocol also does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShare}}$ using execution operations $\mathcal{X}$).
3. Lastly, we show that $\mathcal{A}'$, using such $\mathcal{A}$ as a subroutine, can 'win' the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}',\mathsf{MAC}}(params.\mathcal{P}.n, params.\mathcal{P}.1^\kappa)$ with non-negligible advantage, which implies that MAC is not an asymptotically UF message authentication scheme.

The Adversary $\mathcal{A}'$ and the Protocol $\overline{\mathsf{AuthBroadcast}}$

$\mathcal{A}'$ works as follows:

1. Assume that $\mathcal{A}'$ is given some values of $n, 1^\kappa$ and has access to oracles $\mathsf{OTag}(\cdot)$ and $\mathsf{OVer}(\cdot,\cdot)$. The $\mathsf{OTag}(\cdot)$ oracle takes a message m as input and returns $\mathsf{MAC}_n(k,m)$, where k is a key chosen uniformly from $\{0,1\}^{|1^\kappa|}$ and unknown to $\mathcal{A}'$. The $\mathsf{OVer}(\cdot,\cdot)$ oracle takes inputs m and tag and returns $\top$ if $\mathsf{MAC}_n(k,m) = tag$ and $\bot$ otherwise.
2. $\mathcal{A}'$ modifies the code for $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ into $\overline{\mathsf{AuthBroadcast}}$. Specifically, line 4 in the 'Broadcast' function of $\mathsf{AuthBroadcast}$ (see Algorithm 20) is replaced by:
$$tag \leftarrow \mathsf{OTag}(m \mid\mid timeSent)$$
$\mathcal{A}'$ also changes line 3 in the 'Receive' function of $\mathsf{AuthBroadcast}$ (see Algorithm 21) to:
$$\textbf{and } \mathsf{OVer}(m \mid\mid timeSent, tag) = \top$$
3. $\mathcal{A}'$ executes $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, where $\mathcal{X}$ is {'Sec-in'}, $params.\mathcal{P}.n = n$, $params.\mathcal{P}.1^\kappa = 1^\kappa$, and $\mathcal{A}$ is a PPT subroutine algorithm (discussed below).
4. $\mathcal{A}'$ searches T for an event $\hat{e}_{\text{R}}$ such that the output of $\hat{e}_{\text{R}}$ is ('Receive', m), yet there is no previous 'Broadcast' event $\hat{e}_{\text{B}}$ where the input is m. If $\mathcal{A}'$ finds such an event, it outputs $T.inp[\hat{e}_{\text{R}}].m \mid\mid T.inp[\hat{e}_{\text{R}}].timeSent, T.inp[\hat{e}_{\text{R}}].tag$; otherwise it outputs a pair of randomly chosen strings $x \xleftarrow{\text{R}} \{0,1\}^n, tag \xleftarrow{\text{R}} \{0,1\}^n$.

The Adversary $\mathcal{A}$

Recall, from Definition 3, that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShare}}$ using execution operations

$\mathcal{X} = \{$'Sec-in'$\}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}_{\text{SecKeyShare}}$ such that for some value of $params \in \{0,1\}^*$ holds:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^\kappa) \tag{14}$$

Where:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \Pr\left[\begin{matrix}\pi_{\mathsf{AuthComRcv}_\infty}(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)\end{matrix}\right] - 2^{-params.\mathcal{P}.n} \tag{15}$$

We show now that Equation 14 implies Equation 16, as stated in Lemma 11 below. Note that the difference between Eq. 14 and Eq. 16 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 11. *Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}_{SecKeyShare}$ satisfying Equation 14 for some $params \in \{0,1\}^*$. Then holds:*

$$\epsilon^{\mathcal{R}_{AuthComRcv_\infty}}_{\mathcal{A},\overline{AuthBroadcast},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^\kappa) \tag{16}$$

Where:

$$\epsilon^{\mathcal{R}_{AuthComRcv_\infty},\mathsf{StepCount}}_{\mathcal{A},\overline{AuthBroadcast},\mathcal{X}}(params) \equiv \Pr\left[\begin{matrix}\pi_{AuthComRcv_\infty}(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{AuthBroadcast}}(params)\end{matrix}\right] - 2^{-params.\mathcal{P}.n} \tag{17}$$

Proof of Lemma 11.

From Equation 14, $\mathcal{A}$ is a PPT algorithm that, for some $params \in \{0,1\}^*$ is able to cause the $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ protocol to correctly receive a message without previously broadcasting that message (with non-negligible advantage). Recall that, according to Alg. 20, the 'Broadcast' function of the protocol outputs the message m, local time $timeSent$, and tag $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \mid\mid timeSent)$, where k is unknown to $\mathcal{A}$ due to secure key sharing (except for only negligible probability, as ensured by $\mathcal{M}$). Also recall that, according to Alg. 21, the 'Receive' function of the protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \mid\mid timeSent) = tag$ and that the local time at the receiver minus the timestamp is $\leq params.\mathcal{P}.\Delta$. This means that, with non-negligible advantage, $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \mid\mid timeSent) = tag$.

The only difference between executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)$ and executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$ is that in $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, messages are authenticated using the OTag and OVer oracles instead of using $\mathsf{MAC}_{params.\mathcal{P}.n}(k,\cdot)$, where k is a key chosen uniformly from $\{0,1\}^{|params.\mathcal{P}.1^\kappa|}$, shared securely among the entities, and unknown to $\mathcal{A}$. But $\mathsf{OTag}(\cdot)$ uses $\mathsf{MAC}_{params.\mathcal{P}.n}$ to compute a tag over a message, $\mathsf{OVer}(\cdot,\cdot)$ uses $\mathsf{MAC}_{params.\mathcal{P}.n}$ to verify that

a tag is correct, and both oracles use the same key chosen uniformly from $\{0,1\}^{|params.\mathcal{P}.1^\kappa|}$. This implies that $\mathcal{A}$ must also be able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{OVer}(m \mid\mid timeSent, tag) = \top$ (and moreover the receiver's local time minus $timeSent$ is $\leq params.\mathcal{P}.\Delta$) with non-negligible advantage, even with only negligible probability of being given 'correct' values of tag and $timeSent$ as outputs of a 'Broadcast' event or the key used by the oracles. Therefore, $\mathcal{A}$ satisfies Equation 16.

□

Completing the proof of Claim 1

We can now complete the proof of Claim 2. That is, we show that adversary $\mathcal{A}'$ using $\mathcal{A}$ wins the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$ game with non-negligible advantage.

For convenience, we restate the claim here:

Claim 1. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}_{\mathrm{SecKeyShare}},\,\mathcal{X}}_{\mathrm{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_\infty}$$

Where $\mathcal{M}_{\mathrm{SecKeyShare}}$ is defined in Sec. 8.2.1 and $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ is described in Sec. 8.1.

Proof. Suppose that $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ under $\mathcal{M}$. Then Equation 14 holds (from Definition 3).

Now suppose that we run the game $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$, where $\mathcal{A}'$ uses the algorithm $\mathcal{A}$ discussed above. In the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$ game (Def. 22), the $\mathsf{OTag}(\cdot)$ oracle uses the algorithm MAC_n to compute tags over messages, the $\mathsf{OVer}(\cdot,\cdot)$ oracle uses the algorithm MAC_n to verify tags over messages, and both oracles use the same key chosen uniformly from $\{0,1\}^{|1^\kappa|}$. These are the oracles that $\mathcal{A}'$ uses in the $\overline{\mathsf{AuthBroadcast}}$ protocol.

By Lemma 11, Equation 16 holds. This means that when $\mathcal{A}'$ runs $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, it has non-negligible advantage to find that T contains an event $\hat{e}_{\mathrm{R}}$ such that the output of $\hat{e}_{\mathrm{R}}$ is ('Receive', m), yet there is no previous 'Broadcast' event $\hat{e}_{\mathrm{B}}$ where the input is m. Whenever this is the case, $\mathcal{A}'$ outputs $T.inp[\hat{e}_{\mathrm{R}}].m \mid\mid T.inp[\hat{e}_{\mathrm{R}}].timeSent, T.inp[\hat{e}_{\mathrm{R}}].tag$. Notice that there was no 'Broadcast' operation that could correspond to sending m with timestamp $timeSent$, which means that there was no such corresponding query to the OTag oracle. Recall that the base function of the $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ requirement is $2^{-params.\mathcal{P}.n}$. This implies $\mathcal{A}'$ has non-negligible advantage to win the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$ game, i.e.:

$$\max\{0, \Pr\left[\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa) = \top\right] - 2^{-params.\mathcal{P}.n}\} \notin Negl(1^\kappa) \tag{18}$$

Therefore, MAC is not asymptotically UF (according to Def. 12 in Appx. A). □

8.4.2 Proof of Claim 2

In this section, we prove Claim 2 of Theorem 1 - that $\mathsf{AuthBroadcast}$ ensures freshness and authentication assuming shared-key initialization and bounded clock drift. Note that this proof is similar to the proof of Claim 2 of Theorem 1 (see Sec. 8.4.1), with some changes due to the bounded clock drift assumption and the freshness requirement. We first restate the claim below.

Claim 2. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined in Sec. 2.3). Let $\Delta = params.\mathcal{P}.\Delta + 2\Delta_{clk}$. Let $\mathcal{M} = (\pi, 0)$, where $\pi = \pi_{\text{SecKeyShare}} \wedge \pi^{\text{Drift}}_{\Delta_{clk}}$. Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$$

Where $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ is described in Sec. 8.1.

We prove Claim 2 by contradiction - namely, by showing that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$, then MAC is not asymptotically UF.

Sketch of the Proof of Claim 2.

We complete the proof in three steps:

1. We first define an algorithm $\mathcal{A}'$ and describe how it works. This includes defining a modified version of $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, called $\overline{\mathsf{AuthBroadcast}}$, which uses OTag and OVer instead of $\mathsf{MAC}_{s.n}(s.k, \cdot)$ to tag and authenticate messages.
2. Then we show that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}$ that for some value of $params$ has non-negligible advantage (for requirement $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ and execution-operations set $\mathcal{X}$) to 'win' against the $\overline{\mathsf{AuthBroadcast}}$ protocol (i.e., the $\overline{\mathsf{AuthBroadcast}}$ protocol also does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X}$).
3. Lastly, we show that $\mathcal{A}'$, using such $\mathcal{A}$ as a subroutine, can 'win' the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}',\mathsf{MAC}}(params.\mathcal{P}.n, params.\mathcal{P}.1^\kappa)$ with non-negligible advantage, which implies that MAC is not an asymptotically UF message authentication scheme.

The Adversary $\mathcal{A}'$ and the Protocol $\overline{\mathsf{AuthBroadcast}}$

$\mathcal{A}'$ works as follows:

1. Assume that $\mathcal{A}'$ is given some values of $n, 1^\kappa$ and has access to oracles $\mathsf{OTag}(\cdot)$ and $\mathsf{OVer}(\cdot,\cdot)$. The $\mathsf{OTag}(\cdot)$ oracle takes a message m as input and returns $\mathsf{MAC}_n(k, m)$, where k is a key chosen uniformly from $\{0,1\}^{|1^\kappa|}$ and unknown to $\mathcal{A}'$. The $\mathsf{OVer}(\cdot,\cdot)$ oracle takes inputs m and tag and returns $\top$ if $\mathsf{MAC}_n(k, m) = tag$ and $\bot$ otherwise.

2. $\mathcal{A}'$ modifies the code for $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ into $\overline{\mathsf{AuthBroadcast}}$. Specifically, line 4 in the 'Broadcast' function of $\mathsf{AuthBroadcast}$ (see Algorithm 20) is replaced by:
$$tag \leftarrow \mathsf{OTag}(m \mid\mid timeSent)$$
$\mathcal{A}'$ also changes line 3 in the 'Receive' function of $\mathsf{AuthBroadcast}$ (see Algorithm 21) to:
$$\textbf{and } \mathsf{OVer}(m \mid\mid timeSent, tag) = \top$$
3. $\mathcal{A}'$ executes $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, where $\mathcal{X}$ is {'Sec-in'}, $params.\mathcal{P}.n = n$, $params.\mathcal{P}.1^{\kappa} = 1^{\kappa}$, and $\mathcal{A}$ is a PPT subroutine algorithm (discussed below).
4. $\mathcal{A}'$ searches T for an event $\hat{e}_{\mathrm{R}}$ such that the output of $\hat{e}_{\mathrm{R}}$ is ('Receive', m), yet there is no previous 'Broadcast' event $\hat{e}_{\mathrm{B}}$ where the input is m and where the 'Broadcast' event happened within $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time of the event $\hat{e}_{\mathrm{R}}$ (where Δ_{clk} is the value assumed in the model $\mathcal{M}$). If $\mathcal{A}'$ finds such an event, it outputs $T.inp[\hat{e}_{\mathrm{R}}].m \mid\mid T.inp[\hat{e}_{\mathrm{R}}].timeSent, T.inp[\hat{e}_{\mathrm{R}}].tag$; otherwise it outputs a pair of randomly chosen strings $x \xleftarrow{\mathbf{R}} \{0,1\}^n, tag \xleftarrow{\mathbf{R}} \{0,1\}^n$.

The Adversary $\mathcal{A}$

Recall, from Definition 3, that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ with negligible advantage under model $\mathcal{M}$ using execution operations $\mathcal{X} =$ {'Sec-in'}, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}$ such that for some value of $params \in \{0,1\}^*$ holds:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{19}$$

Where $\Delta = params.\mathcal{P}.\Delta + 2\Delta_{clk}$ and:

$$\begin{aligned}&\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \\ &\quad \Pr\left[\begin{matrix}\pi_{\mathsf{AuthComRcv}_{\Delta}}(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)\end{matrix}\right] - 2^{-params.\mathcal{P}.n}\end{aligned} \tag{20}$$

We show now that Equation 19 implies Equation 21, as stated in Lemma 12 below. Note that the difference between Eq. 19 and Eq. 21 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 12. *Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}$ satisfying Equation 19 for some $params \in \{0,1\}^*$. Then holds:*

$$\epsilon^{\mathcal{R}_{AuthComRcv_{\Delta}}}_{\mathcal{A},\overline{AuthBroadcast},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{21}$$

Where $\Delta = params.\mathcal{P}.\Delta + 2\Delta_{clk}$ and:

$$\begin{aligned}&\epsilon^{\mathcal{R}_{AuthComRcv_{\Delta}},\mathsf{StepCount}}_{\mathcal{A},\overline{AuthBroadcast},\mathcal{X}}(params) \equiv \\ &\quad \Pr\left[\begin{matrix}\pi_{AuthComRcv_{\Delta}}(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{AuthBroadcast}}(params)\end{matrix}\right] - 2^{-params.\mathcal{P}.n}\end{aligned} \tag{22}$$

Proof of Lemma 12.

From Equation 19, $\mathcal{A}$ is a PPT algorithm that, for some $params \in \{0,1\}^*$ is able to cause the $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ protocol to correctly receive a message without previously broadcasting that message within the last $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time (with non-negligible advantage). Recall that, according to Alg. 20, the 'Broadcast' function of the protocol outputs the message m, local time $timeSent$, and tag $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent)$, where k is unknown to $\mathcal{A}$ due to secure key sharing (except for only negligible probability, as ensured by $\mathcal{M}$). Also recall that, according to Alg. 21, the 'Receive' function of the protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) = tag$ and that the local time at the receiver minus the timestamp is $\leq params.\mathcal{P}.\Delta$. This means that $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) = tag$. But since $\mathcal{M}$ ensures that local time is always within Δ_{clk} of the real time (except with negligible probability), this implies that, except with negligible probability, 'correct' values of the tag and timestamp (i.e., values that would allow the message to be received) are only output at such 'Broadcast' events where the real time of the 'Broadcast' event is within $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ of the real time of the 'Receive' event. Therefore, except with negligible probability, $\mathcal{A}$ is not given such 'correct' values or the key k, yet, with non-negligible advantage, $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) = tag$ and the receiver's local time minus $timeSent$ is $\leq params.\mathcal{P}.\Delta$.

The only difference between executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)$ and executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$ is that in $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, messages are authenticated using the OTag and OVer oracles instead of using $\mathsf{MAC}_{params.\mathcal{P}.n}(k, \cdot)$, where k is a key a key chosen uniformly from $\{0,1\}^{|params.\mathcal{P}1^\kappa|}$, shared securely among the entities, and unknown to $\mathcal{A}$. But $\mathsf{OTag}(\cdot)$ uses $\mathsf{MAC}_{params.\mathcal{P}.n}$ to compute a tag over a message, $\mathsf{OVer}(\cdot,\cdot)$ uses $\mathsf{MAC}_{params.\mathcal{P}.n}$ to verify a tag over message, and both oracles use the same key chosen uniformly from $\{0,1\}^{|params.\mathcal{P}1^\kappa|}$. This implies that $\mathcal{A}$ must also be able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{OVer}(m \,||\, timeSent, tag) = \top$ and the receiver's local time minus $timeSent$ is $\leq params.\mathcal{P}.\Delta$ with non-negligible advantage, even with only negligible probability of being given 'correct' values of tag and $timeSent$ as outputs of a 'Broadcast' event or the key used by the oracles. Therefore, $\mathcal{A}$ satisfies Equation 21.

□

Completing the proof of Claim 2

We can now complete the proof of Claim 2. That is, we show that adversary $\mathcal{A}'$ using $\mathcal{A}$ wins the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$ game with non-negligible advantage.

For convenience, we restate the claim here:

Claim 2. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation

is defined in Sec. 2.3). Let $\Delta = params.\mathcal{P}.\Delta + 2\Delta_{clk}$. Let $\mathcal{M} = (\pi, 0)$, where $\pi = \pi_{\text{SecKeyShare}} \wedge \pi^{\text{Drift}}_{\Delta_{clk}}$. Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$$

Where $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ is described in Sec. 8.1.

Proof. Suppose that $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ under $\mathcal{M}$. Then Equation 19 holds (from Definition 3).

Now suppose that we run the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$, where $\mathcal{A}'$ uses the algorithm $\mathcal{A}$ discussed above. In the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}',\mathsf{MAC}}(n, 1^\kappa)$ game (Def. 22), the $\mathsf{OTag}(\cdot)$ oracle uses the algorithm MAC_n to compute tags over a message, the $\mathsf{OVer}(\cdot,\cdot)$ oracle uses the algorithm MAC_n to verify tags over messages, and both oracles use the same key chosen uniformly from $\{0,1\}^{|1^\kappa|}$. These are the oracles that $\mathcal{A}'$ uses in the $\overline{\mathsf{AuthBroadcast}}$ protocol.

By Lemma 12, Equation 21 holds. This means that when $\mathcal{A}'$ runs $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, it has non-negligible advantage to find that T contains an event $\hat{e}_{\text{R}}$ such that the output of $\hat{e}_{\text{R}}$ is $(\text{`Receive'}, m)$, yet there is no previous 'Broadcast' event $\hat{e}_{\text{B}}$ where the input is m and where the 'Broadcast' event happened within $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time of the event $\hat{e}_{\text{R}}$. Whenever this is the case, $\mathcal{A}'$ outputs $T.inp[\hat{e}_{\text{R}}].m \,||\, T.inp[\hat{e}_{\text{R}}].timeSent, T.inp[\hat{e}_{\text{R}}].tag$. Notice that there was no 'Broadcast' operation that could correspond to sending m with timestamp $timeSent$, which means that there was no such corresponding query to the OTag oracle. Recall that the base function of the $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ requirement is $2^{-params.\mathcal{P}.n}$. This implies $\mathcal{A}'$ has non-negligible advantage to win the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$ game, i.e.:

$$\max\{0, \Pr\left[\mathbf{Exp}^{\text{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa) = \top\right] - 2^{-params.\mathcal{P}.n}\} \notin Negl(1^\kappa) \tag{23}$$

Therefore, MAC is not asymptotically UF (according to Def. 12 in Appx. A). □

8.4.3 Proof of Claim 3

In this section, we prove Claim 3 of Theorem 1 - that $\mathsf{AuthBroadcast}$ ensures bounded-delay delivery of broadcast messages assuming $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$. The claim is restated below.

Claim 3. Let MAC be a message authentication scheme, as defined in Appendix A. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined in Sec. 2.3). Then:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}},\,\mathcal{X}}_{\text{poly}} \mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$$

Where $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ is defined in Sec. 8.2.2 and $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ is defined in Sec. 8.1.

We prove Claim 3 by contradiction - we show that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ with negligible advantage under model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ using execution operations $\mathcal{X}$, then MAC does not satisfy the definition of message authentication scheme.

Proof. From Definition 3, if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ with negligible advantage under model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ using execution operations $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ such that for some $params \in \{0,1\}^*$ holds:

$$\epsilon^{\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{24}$$

Where

$$\epsilon^{\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \Pr\left[\begin{matrix}\pi_{\mathsf{Broadcast}_{\Delta_{com}}}(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)\end{matrix}\right] \tag{25}$$

That is, the adversary $\mathcal{A}$ is able to prevent the successful reception of a broadcast message with non-negligible probability.

However, the model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ ensures that, except for negligible probability, if the output of an event is $(\text{'}\mathsf{Broadcast}\text{'}, m, timeSent, tag)$ (i.e., some entity i broadcasts a message), then for every other entity j, there is a later 'Receive' event at j within Δ_{com} real time where $(m, timeSent, tag)$ is received as input, (see Alg. 15) and $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}.\Delta$ (see Algorithm 16).

Recall that, according to Alg. 20, the 'Broadcast' function of $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, for input message m, outputs $(\text{'}\mathsf{Broadcast}\text{'}, m, timeSent, \mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent))$, where $params.\mathcal{P}.n$ is the length of tags used, $timeSent$ is the local time, and k is a key shared securely among all entities (except for negligible probability, as ensured by $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$). Also recall that, according to Alg. 21, the 'Receive' function of the $\mathsf{AuthBroadcast}$ protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) = tag$ and that the local time at the receiver minus the timestamp is $\leq params.\mathcal{P}.\Delta$. If this holds, the 'Receive' function outputs $(\text{'}\mathsf{Receive}\text{'}, m)$.

Thus, since $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$, then, with overwhelming probability, for every 'Broadcast' event with input m at entity i, for every other entity j, there is a later 'Receive' event at j within Δ_{com} real time where $(m, timeSent, \mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent))$ is received as input. Since $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ ensures that $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}.\Delta$ and that local time is always within Δ_{clk} of the real time (except with negligible probability), then the local time at the receiver minus the timestamp is $\leq params.\mathcal{P}.\Delta$. Consequently, if some message is broadcast, yet the message is not successfully received, then there must be some nonzero probability that the reason is that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) \neq \mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent)$ (in the 'Receive' function), which means that MAC returns different values when evaluated multiple times on the same inputs, which means that it is not a deterministic function. This contradicts Definition 11 in Appx. A, implying that MAC is not a message authentication scheme. □

9 Conclusions and Future Work

The MoSS framework allows analysis of applied cryptographic protocols, with different communication, synchronization and adversary models. Security in MoSS

is defined with respect to the execution process in Algorithm 1, and given model and requirement specifications. MoSS allows comparison of protocols based on the requirements they satisfy and the models they assume. Definitions of models and requirements may be reused across different protocols and schemes.

Future work includes the important challenges of (1) extending the MoSS framework to support secure compositions, as in UC, and (2) using computer-aided proofs with the framework.

References

1. Backes, M., Pfitzmann, B., Waidner, M.: A general composition theorem for secure reactive systems. In: Theory of Cryptography Conference. Springer (2004)
2. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: Sok: Computer-aided cryptography. IACR Cryptol. ePrint Arch. (2019)
3. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Annual Cryptology Conference (2011)
4. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: EUROCRYPT (2006)
5. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing (2008)
6. Bowman, H., Gomez, R.: Concurrency theory: calculi an automata for modelling untimed and timed concurrent systems. Springer Science & Business Media (2006)
7. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible universal composability made simple. In: EUROCRYPT (2019)
8. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: IEEE Symposium on Foundations of Computer Science (2001)
9. Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: CRYPTO (2015)
10. CCITT, B.B.: Recommendations X. 509 and ISO 9594-8. Information Processing Systems-OSI-The Directory Authentication Framework (Geneva: CCITT) (1988)
11. Hofheinz, D., Shoup, V.: Gnuc: A new universal composability framework. Journal of Cryptology **28**(3), 423–508 (2015)
12. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The theory of timed i/o automata. Synthesis Lectures on Distributed Computing Theory **1**(1), 1–137 (2010)
13. Koblitz, N., Menezes, A.: Critical perspectives on provable security: Fifteen years of" another look" papers. Advances in Mathematics of Communications (2019)
14. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology pp. 1–124 (2020)
15. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Jun 2013). https://doi.org/10.17487/RFC6962, https://rfc-editor.org/rfc/rfc6962.txt
16. Leibowitz, H., Herzberg, A., Syta, E.: Provably Secure PKI Schemes. Cryptology ePrint Archive, Report 2019/807 (2019), https://eprint.iacr.org/2019/807
17. Leibowitz, H., Piotrowska, A.M., Danezis, G., Herzberg, A.: No Right to Remain Silent: Isolating Malicious Mixes. In: USENIX Security 19 (2019)
18. Lochbihler, A., Sefidgar, S.R., Basin, D., Maurer, U.: Formalizing constructive cryptography using crypthol. In: Computer Security Foundations (2019)
19. Maurer, U.: Constructive cryptography–a new paradigm for security definitions and proofs. In: Workshop on Theory of Security and Applications (2011)

20. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Computer Aided Verification (2013)
21. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
22. Wikström, D.: Simplified universal composability framework. In: Theory of Cryptography Conference. Springer (2016)

A Message Authentication Scheme

In this section, we provide definitions for *message authentication scheme* and *unforgeable* message authentication scheme.

Definition 11. *A message authentication scheme* MAC *is a deterministic algorithm* $\mathsf{MAC}_n(k, m) \rightarrow tag$*, with inputs tag length* n*, shared key* k*, and message* m*, and output* n*-bit tag* tag*.*

Definition 12 (Asymptotically Unforgeable). *We call a message authentication scheme* MAC asymptotically unforgeable (asymptotically UF) *if for every PPT adversary* $\mathcal{A}$*, tag length* n*, and* 1^κ*, the maximum of 0 and probability that* $\mathcal{A}$ *'wins' the game* $\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$ *minus* 2^{-n} *is negligible in* 1^κ*, i.e.:*

$$\forall\ \mathcal{A} \in PPT, n \in \mathbb{N}, 1^\kappa : \\ max\{0, Pr[\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa) = \top] - 2^{-n}\} \in Negl(1^\kappa) \tag{26}$$

where $\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$ *is defined in Algorithm 22. In the opposite case, if there exists an adversary* $\mathcal{A}$ *such that* $max\{0, Pr[\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa) = \top] - 2^{-n}\} \notin Negl(1^\kappa)$ *for some* $n, 1^\kappa$*, we say that* $\mathcal{A}$ wins the game with non-negligible advantage*.*

Algorithm 22 $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$

$k \xleftarrow{R} \{0,1\}^{|1^\kappa|}$
$S = \emptyset$
define: $\mathsf{OTag}(m)$: $tag \leftarrow \mathsf{MAC}_n(k, m)$; $S \leftarrow S \cup \{m\}$; **return** tag
define: $\mathsf{OVer}(m, tag)$: **if** $\mathsf{MAC}_n(k, m) = tag$ **then return** $\top$ **else return** $\bot$
$m', tag' \leftarrow \mathcal{A}^{\mathsf{OTag}(\cdot),\mathsf{OVer}(\cdot,\cdot)}(n, 1^\kappa)$
if $m' \notin S$ **and** $\mathsf{OVer}(m', tag') = \top$ **then return** $\top$
else return $\bot$
end if

B Additional Specification Predicates

B.1 Model Predicates

B.1.1 $\pi^{|\mathsf{F}|\leq f}$: up to f Byzantine (Malicious) Faults

We next define $\pi^{|\mathsf{F}|\leq f}$, a specific adversary model predicate allowing the adversary to choose, and completely control, up to f of the entities in N. We refer to such failures, where the adversary is allowed to completely control the entity, as *malicious* or *Byzantine* faults. We use f as a function applied to the total number of entities $|\mathsf{N}|$. We refer to this particular faults model as $\mathcal{M}^{|\mathsf{F}|\leq f}$, where $f : \mathbb{N} \to \mathbb{N}$ bounds the number of faulty entities as a function of the total number of entities. Specifically, the adversary may corrupt entities by performing the 'Get-state', 'Set-state', and 'Set-output' operations of the set $\mathcal{X}$. To enforce the model predicate, we simply ensure that the 'Get-state', 'Set-state', and 'Set-output' operations can be applied only to entities in $T.\mathsf{F}$, and that $|T.\mathsf{F}| \leq f(|\mathsf{N}|)$. We define the $\pi^{|\mathsf{F}|\leq f}$ predicate in Algorithm 23.

Algorithm 23 $\pi^{|\mathsf{F}|\leq f}$ (T) Predicate

```
1: return (
2:     (|T.F| ≤ f(|T.N|))                                  ▷ Max size of T.F is not exceeded
3:     and ∀ê ∈ {1, ..., T.e} :                            ▷ For each event
4:         if T.opr[ê] ∈ {'Get-state', 'Set-state', 'Set-output'} and
               T.type[ê] = 'X'                             ▷ If the operation means the adversary controls the entity
5:         then T.ent[ê] ∈ T.F                             ▷ Then entity is in T.F
    )
```

B.1.2 $\pi^{\mathbf{AuthCom}}_{\boldsymbol{\Delta_{com}}}$: authentic-sender, bounded-delay communication

We next present $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$, an authentic-sender, bounded-delay communication model predicate. It is convenient to define $\pi^{\text{AuthCom}}_{\Delta_{com}}$ as a conjunction of two simpler predicates: $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$, ensuring authentic-sender for message-receive events, and $\pi^{\text{Com}}_{\Delta_{com}}$, ensuring reliable, bounded-delay for message-send events. Namely:

$$\pi^{\text{AuthCom}}_{\Delta_{com}}(T) = \pi^{\text{Com}}_{\Delta_{com}}(T) \wedge \pi^{\text{AuthCom-rcv}}_{\Delta_{com}}(T) \tag{27}$$

We first present $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$, which ensures authentic-sender for message-receive events. The adversary decides on the function $opr[\hat{e}_{\text{R}}]$ to be invoked at every event $\hat{e}_{\text{R}}$ as well as the input $inp[\hat{e}_{\text{R}}]$. We assume a convention for *send* and *receive* events as follows. The adversary causes a message receipt event by setting $opr[\hat{e}_{\text{R}}]$ to 'Receive' and $inp[\hat{e}_{\text{R}}]$ to (m, i_{S}) (where m is the message and $i_{\text{S}} \in \mathsf{N}$ is the purported sender). We use dot notation to refer to the message ($inp[\hat{e}_{\text{R}}].m$) and to the sender ($inp[\hat{e}_{\text{R}}].i_{\text{S}}$). Also, we allow the sender $ent[\hat{e}_{\text{S}}]$ to specify, as part

of its output $out[\hat{e}_{\mathrm{S}}]$, one or more triplets of the form $(\text{'send'}, m, i_{\mathrm{R}})$, indicating the sending of message m to $i_{\mathrm{R}} \in \mathsf{N}$.

The authentic-sender property ($\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate) implies that $inp[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ indeed sent this message to $ent[\hat{e}_{\mathrm{R}}]$, during some previous event $\hat{e}_{\mathrm{S}} < \hat{e}_{\mathrm{R}}$. The $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate is shown in Algorithm 24.

Algorithm 24 $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ (T) Predicate

return (
  $\forall \hat{e}_{\mathrm{R}} \in \{1, \ldots, T.e\}$:
    if $T.opr[\hat{e}_{\mathrm{R}}] =$ 'Receive': ▷ *For each message-receive event*
    and $T.ent[\hat{e}_{\mathrm{R}}], T.inp[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}} \in T.\mathsf{N} - T.\mathsf{F}$ ▷ *If both receiver and purported sender are honest*
    then $\exists \hat{e}_{\mathrm{S}} \in \{1, \ldots, \hat{e}_{\mathrm{R}} - 1\}$ ▷ *Then there is a previous event*
      s.t. $(\text{'send'}, T.inp[\hat{e}_{\mathrm{R}}].m, T.ent[\hat{e}_{\mathrm{R}}]) \in T.out[\hat{e}_{\mathrm{S}}]$ ▷ *In which an entity sent the message to the receiver*
      and $T.ent[\hat{e}_{\mathrm{S}}] = T.inp[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ ▷ *And that entity was the purported sender*
)

The $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate ensures reliable, bounded-delay delivery of messages sent. Assume that at event $\hat{e}_{\mathrm{S}}$ of the execution, the output $out[\hat{e}_{\mathrm{S}}]$ generated by $ent[\hat{e}_{\mathrm{S}}]$, includes a $(\text{'send'}, m, j)$ triplet, i.e., $ent[\hat{e}_{\mathrm{S}}]$ sends message m to $j \in \mathsf{N}$. If the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate is true for this execution, then after at most Δ_{com}, if the execution did not terminate already, then entity j would receive m from $ent[\hat{e}_{\mathrm{S}}]$. The $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate is shown in Algorithm 25.

Algorithm 25 $\pi^{\text{Com}}_{\Delta_{com}}$ (T) Predicate

return (
  $\forall \hat{e}_{\mathrm{S}} \in \{1, \ldots, T.e - 1\}$:
    if ($\exists(\text{'send'}, m, i_{\mathrm{R}}) \in T.out[\hat{e}_{\mathrm{S}}]$ ▷ *If the output includes a send triple*
    and $T.\tau[T.e] \geq T.\tau[\hat{e}_{\mathrm{S}}] + \Delta_{com}$ ▷ *And execution did not terminate yet after Δ_{com} real time*
    and $T.ent[\hat{e}_{\mathrm{S}}] \in T.\mathsf{N} - T.\mathsf{F}$) ▷ *And the entity is honest*
    then $\exists \hat{e}_{\mathrm{R}} \in \{\hat{e}_{\mathrm{S}} + 1, \ldots, T.e\}$ ▷ *Then there is a later event*
      s.t. $T.\tau[\hat{e}_{\mathrm{S}}] + \Delta_{com} \geq T.\tau[\hat{e}_{\mathrm{R}}]$ ▷ *Within Δ_{com} real time*
      and $T.ent[\hat{e}_{\mathrm{R}}] = i_{\mathrm{R}}$ ▷ *Where the entity is the intended recipient in the send triple*
      and $T.opr[\hat{e}_{\mathrm{R}}] =$ 'Receive' ▷ *And which is a receive event*
      and $T.inp[\hat{e}_{\mathrm{R}}] = (m, T.ent[\hat{e}_{\mathrm{S}}])$ ▷ *And in which the entity receives the message from the sender*
)

We remark that: $\pi^{\text{AuthCom}}_{\Delta_{com}}$ only applies when both sender and recipient are honest (i.e., in $\mathsf{N}-\mathsf{F}$); $\pi^{\text{AuthCom}}_{\Delta_{com}}$ only ensures delivery, sender authentication and bounded delay. This still allows receipt of duplicate messages, which may involve unbounded delay. To simplify $\pi^{\text{Com}}_{\Delta_{com}}$, we use the adversary-controlled $\tau[\cdot]$ values (line 6 of Algorithm 1). For this to be meaningful, we depend on the synchronization properties of the $\pi^{\text{CLK}}_{\Delta_{clk}}$ model predicate, discussed next.

B.1.3 $\pi^{\text{CLK}}_{\Delta_{clk}}$: bounded-drift clock synchronization assumptions

We present $\pi^{\text{CLK}}_{\Delta_{clk}}$, which models the bounded-drift clock synchronization assumptions. We split this into two predicates: $\pi^{\text{Drift}}_{\Delta_{clk}}$, which limits the drift between the clock values $clk[\hat{e}]$ (provided by the adversary as input to the protocol) and the real time values $\tau[\hat{e}]$; and $\pi^{\text{Wake-up}}_{\Delta_{clk}}$, which provides a 'wake-up service' to the protocol. Namely:

$$\pi^{\text{CLK}}_{\Delta_{clk}}(T) = \pi^{\text{Drift}}_{\Delta_{clk}}(T) \wedge \pi^{\text{Wake-up}}_{\Delta_{clk}}(T) \tag{28}$$

$\pi^{\text{Drift}}_{\Delta_{clk}}$ is presented in Algorithm 3.

$\pi^{\text{Wake-up}}_{\Delta_{clk}}$ provides a 'wake-up service' allowing the protocol to perform time-driven activities and ensuring that appropriate functions are invoked properly. This is ensured by requiring that if $(\text{'}Sleep\text{'}, x)$ was part of the output $out[\hat{e}]$ (indicating that entity $ent[\hat{e}]$ was 'put to sleep' for x time) and execution did not terminate by 'real' time $\tau[\hat{e}] + x + \Delta_{clk}$, then at some event $\hat{e}' > \hat{e}$ (where $\tau[\hat{e}']$ was within Δ_{clk} from $\tau[\hat{e}] + x$), the same entity ($ent[\hat{e}]$) was indeed 'Woken up'. The $\pi^{\text{Wake-up}}_{\Delta_{clk}}$ predicate appears in Algorithm 26.

Algorithm 26 $\pi^{\text{Wake-up}}_{\Delta_{clk}}(T)$ Predicate

```
1: return (
2:   ∀ê ∈ {1, ..., T.e}:                          ▷ For each event ê
3:     if ( ('Sleep', x) ∈ T.out[ê]                ▷ If the output includes a ('Sleep', x) tuple
4:          and T.τ[T.e] ≥ T.τ[ê]+x+Δ_clk )        ▷ And execution did not terminate yet after x + Δ_clk real time
5:     then ∃ê' ∈ {ê + 1, ..., T.e}                ▷ Then there is a later event
6:          s.t. |T.τ[ê'] − T.τ[ê] − x| ≤ Δ_clk    ▷ With real time x greater than at ê (within Δ_clk)
7:          and T.ent[ê'] = T.ent[ê]               ▷ In which the entity is the same as in ê
8:          and T.opr[ê'] = 'Wake-up'              ▷ And the operation is 'Wake-up'
   )
```

B.1.4 $\pi^{\hat{r}\text{-rounds}}_{\text{SecInit}}$: the $\hat{r}$-rounds Secure Initialization

Cryptographic protocols are often designed assuming a secure initialization process, e.g., assuming shared secret keys. However, in the execution process (Algorithm 1), entities can only communicate via the adversary. As a result, we

cannot simply *assume* shared secret keys, but the entities can use their local randomness to generate secret keys, and they can communicate, using cryptography, to securely establish shared secret values. We next define a simple secure initialization model predicate, $\pi^{\hat{r}\text{-rounds}}_{\text{SecInit}}$. This model predicate ensures $\hat{r}$ secure 'rounds' of $|\mathsf{N}|$ events each, where in event $\hat{e}$ (where $1 \leq \hat{e} \leq \hat{r} \cdot |\mathsf{N}|$) holds:

- Entities are invoked with the special operation 'Init', i.e., $opr[\hat{e}] =$ 'Init', and in 'round robin', i.e., $ent[\hat{e}] = \hat{e} \mod |\mathsf{N}|$ (where $\mathsf{N} = \{1, 2, \ldots\}$). Note, in particular, that this prevents the adversary, during the initialization, from invoking the special 'Set-state', 'Set-output', and 'Get-state' operations to control the state or output of an entity ('Set-state' or 'Set-output') or to expose the state of an entity ('Get-state').
- Authenticated, reliable communication. Namely, every message received by entity i_{R} from entity i_{S} at round $2 \leq r \leq \hat{r}$, was indeed sent by i_{S} in the previous round to i_{R}; and vice verse, i.e., every message sent by i_{S} to i_{R} at round $1 \leq r \leq (\hat{r} - 1)$, is correctly received by i_{R}, from sender i_{S}, in the next round.

It is convenient to capture each of these two aspects by a separate model predicate, i.e.:

$$\pi^{\hat{r}\text{-rounds}}_{\text{SecInit}}(T) = \pi^{\hat{r}\text{-rounds}}_{\text{InitOps}}(T) \wedge \pi^{\hat{r}\text{-rounds}}_{\text{InitCom}}(T) \tag{29}$$

where $\pi^{\hat{r}\text{-rounds}}_{\text{InitOps}}$ captures the first aspect ('operations') and $\pi^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ captures the second aspect ('communications'). We now define each of these more precisely; for convenience, let $\mathsf{N} = \{1, 2, \ldots\}$. The $\pi^{\hat{r}\text{-rounds}}_{\text{InitOps}}$ predicate is shown in Algorithm 27.

Algorithm 27 $\pi^{\hat{r}\text{-rounds}}_{\text{InitOps}}(T)$ Predicate

```
1: return (
2:   ∀i ∈ T.N, r ∈ {0, ..., r̂ − 1} :        ▷ For each entity i and each round r
3:       (T.ent[i + r · |T.N|] = i)          ▷ 'Round robin'
4:       and (T.opr[i + r · |T.N|] = 'Init')  ▷ Operation is 'Init'
     )
```

To define the $\pi^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ (sub)model predicate, we assume the following conventions. To cause an entity to receive a message, the adversary includes the triple ('auth_init_recv', m, i_{S}) (where m is the message and $i_{\text{S}} \in \mathsf{N}$ is the purported sender) as part of the input $inp[\hat{e}_{\text{R}}]$. The sender indicates the sending of message m to $i_{\text{R}} \in \mathsf{N}$ by specifying, as part of its output $out[\hat{e}_{\text{S}}]$, a triplet ('auth_init_send', m, i_{R}). The $\pi^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ predicate is shown in Algorithm 28.

Algorithm 28 $\pi^{\hat{r}\text{-rounds}}_{\text{InitCom}}(T)$ Predicate

```
1: return (
2:     ∀i_S, i_R ∈ T.N,  r ∈ {0, ..., r̂ − 2},  m ∈ {0,1}*  :      ▷ For pairs of entities in T.N, for each round except the last one, and for any message m
3:         ('auth_init_send', m, i_R) ∈ T.out[i_S + r′ · |T.N|]      ▷ i_S sends m to i_R
4:         if and only if
5:         ('auth_init_recv', m, i_S) ∈ T.inp[i_R + (r′+1) · |T.N|]  ▷ i_R receives m from i_S in the next round
       )
```

B.2 Requirement Predicates

B.2.1 Verifiable Attribution of Statements (VAS)

The output of many protocols may include *attributable statements*. An *attributable statement* is a tuple (m, σ, i), where m is a string, $i \in \mathsf{N}$ is the *purported origin* of the statement, and σ provides *evidence* (typically, a signature), allowing attribution of statement m to entity i. We next explain the *validation* process, which uses the evidence σ to establish if i has, in fact, originated m.

We focus on the typical case, where attribution is based on the use of a *digital signature scheme* $\mathcal{S}$, applied by the protocol $\mathcal{P}$. Namely, σ is the result of applying the signing algorithm $\mathcal{S}.Sign$ to the message m, using some (private) signing key sk belonging to the origin i. Therefore, we say that the attributable statement (m, σ, i) is *valid*, i.e., that σ really 'proves' that i is the origin of m, if $\mathcal{S}.\mathsf{Ver}(pk, m, \sigma) = \top$, where pk is the public signature-verification key of i, i.e., the public key that validates signatures computed using sk. This *attributes* the message m to the 'owner' of the public key pk (and the corresponding signing key sk). To attribute m to i, it remains to establish the association between i and the public key pk, i.e., to attribute pk, and messages verified by it, to i. We focus on protocols where this association is known and secure ('off-band'), e.g., CA public keys in PKI schemes.

We formalize this by assuming that each entity $i \in \mathsf{N}$ *identifies* its public key pk by outputting the pair $(\text{'public key'}, pk) \in out[\hat{e}]$, in some event $\hat{e}$; namely, we use 'public key' as a 'label', to identify output of the public key. Typically, entities output the public key when they generate the key, i.e., $ent[\hat{e}] = i$, possibly as an initialization operation, i.e. $opr[\hat{e}] =$ *'Init'*. Notice that entities may often also send their public keys to each other using the $(\text{'send'}, m, i_{\mathrm{R}})$ output convention described in § B.1.2; however, we prefer to keep the two conventions separate, since we believe that not every protocol that uses verification of attribution would necessarily send public keys in precisely the same way.

More precisely, the following *Key Attribution Predicate* V_{ka} outputs $\top$ if entity i has identified pk as its public key in a given transcript T output by an execution of the protocol $\mathcal{P}$ (Algorithm 1):

$$V_{ka}(i, pk, T) = \{\exists \hat{e} \ s.t. \ T.ent[\hat{e}] = i \ \wedge (\text{'public key'}, pk) \in T.out[\hat{e}]\} \qquad (30)$$

We now define the Verifiable Attribution of Statements requirement predicate π_{VAS}. The adversary $\mathcal{A}$ 'wins' in the experiment if its output $out_{\mathcal{A}}$ includes both a valid attributable statement (m, σ, i) for non-faulty entity $i \in \mathsf{N} - \mathsf{F}$ and a verification key pk associated with i, yet i did *not* originate m. To allow us to identify events $\hat{e}$ in which an entity $i = ent[\hat{e}]$ intentionally signed message m, we adopt the following convention: whenever signing a message m, the party adds the pair $(\text{'signed'}, m)$ as part of its output, i.e., $(\text{'signed'}, m) \in out[\hat{e}]$. Since this is *always* done, whenever the protocol signs a message, we will *not* explicitly include the $(\text{'signed'}, m)$ pairs as part of the output, which would make the pseudo-code cumbersome. Note that often the entity will also send the signed message, however, different protocols may send in different ways, hence this convention makes it easier to define the requirement predicate.

The requirement predicate π_{VAS} is defined with respect to specific signature scheme $\mathcal{S}$, and the V_{ka} predicate defined above (Eq. 30). For simplicity, and since $\mathcal{S}$ is typically obvious (as part of $\mathcal{P}$), we do not explicitly specify $\mathcal{S}$ as a parameter of the requirement predicate. The π_{VAS} predicate is shown in Algorithm 29.

Algorithm 29 Verifiable Attribution of Statements Predicate $\pi_{\mathsf{VAS}}(T)$

```
1: (m, σ, i, pk) ← T.out_A
2: return ¬(
3:     i ∈ T.N − T.F                                  ▷ i is an honest entity
4:     and S.Ver(pk, m, σ) = ⊤                        ▷ m was signed by the owner of pk
5:     and V_ka(i, pk, T) = ⊤                         ▷ i identified pk as its public key
6:     and ∄ê s.t.: T.ent[ê] = i
           and ('signed', m) ∈ T.out[ê]               ▷ Yet, i never indicated that it signed m
       )
```

B.2.2 Generic Misbehavior Detection

Many security protocols are required to be *resilient to misbehaviors*, i.e., to achieve their goals even if some of the entities, say entities in $\mathsf{F} \subset \mathsf{N}$, are faulty, and may misbehave (arbitrarily or in some specified manner). This resiliency to faulty, misbehaving entities is often based on *detection of misbehavior*; furthermore, often, many security protocols are required only to *detect misbehaviors*, which would be followed by taking some additional measures to deter and/or neutralize an attack. While misbehavior can be detected in different ways, detection is typically based either on some *evidence* that a certain entity is dishonest, where the evidence should be *verifiable by any third party*, or based on an *accusation*, where one entity (the *accuser*) accuses another entity (the *suspect*) of some misbehavior. Such an accusation may not be true, and therefore, it is harder to use this approach to deter and/or neutralize the attack; however, many misbehaviors do not leave any *evidence* verifiable by a third party, in which case, accusations may provide some security benefits, e.g., *detection* of the attack. A

typical example of such misbehavior that does not leave any evidence is when a party *fails to act* in a required way, e.g., to send a required message or response; such failure may be plausibly blamed on communication issues, or on failure of the intended recipient. Often, a party, say Alice, detects such failure, say of Mal, to send a required message, after Alice waits for some *maximum delay*, and then Alice issues an 'accusation' against Mal, to alert others; for example, see [17]. An honest entity would only accuse a misbehaving party; however, because an accusation cannot be verified, a misbehaving entity could falsely accuse anyone, even an honest entity.

To formalize these concepts, we define two requirement predicates: one to ensure that honest entities cannot be 'framed' as misbehaving, i.e., evidences are always verifiable with correct outcome, and another one to express that honest entities never accuse other honest entities, i.e., only accuse misbehaving entities.

The Non-frameability requirement and Proof of Misbehavior. The first security requirement predicate is called *non-frameability* (of honest entities), and ensures that a specific protocol would not allow any entity to produce a *valid Proof of Misbehavior* of a non-faulty entity. The requirement predicate is therefore defined with respect to a given *Proof of Misbehavior Validation Predicate* V_{PoM}, which receives two inputs: a Proof-Validation Key pk and a purported-proof ζ. The output of $V_{PoM}(pk, \zeta)$ is $\top$ if and only if ζ is a valid Proof of Misbehavior, as indicated by pk; i.e., a misbehavior by an entity who knows the corresponding private key, typically, the 'owner' of pk, which can be validated using the Key Attribution Predicate V_{ka}. The natural way is to define the Proof of Misbehavior Validation Predicate V_{PoM} to be *protocol specific*, as the notions of misbehavior, and valid proof of misbehavior, depend on the specific protocol specifications. We specify for $\mathcal{P}$ a special *stateless* operation $opr =$'V_{PoM}', which does not modify the state or depend on it, or on the local clock. Abusing notation, we denote this operation simply as $\mathcal{P}.V_{PoM}(pk, \zeta)$. The use of a protocol-defined $\mathcal{P}.V_{PoM}$ allows us to define, below, the Non-frameability requirement predicate.

Let $V_{PoM} : \{0,1\}^* \times \{0,1\}^* \to \{\top, \bot\}$ be a predicate. The Non-frameability predicate π_{NF}, shown in Algorithm 30, returns $\bot$ if the adversary was able to output a Proof of Misbehavior for an honest entity, and $\top$ otherwise.

Algorithm 30 Non-frameability Predicate $\pi_{\mathsf{NF}}(T)$

1: $(i, \zeta, pk) \leftarrow T.out_{\mathcal{A}}$
2: **return** $\neg($
3: $\quad V_{ka}(i, pk, T)$ ▷ *i identified pk as its public key*
4: $\quad$ **and** $\mathcal{P}.V_{PoM}(pk, \zeta)$ ▷ *ζ is a valid Proof of Misbehavior by the owner of pk*
5: $\quad$ **and** $i \in T.\mathsf{N} - T.\mathsf{F}$ ▷ *i is an honest entity*
$\quad)$

B.2.3 Accusations and the No False Accusations Predicate

Recall that in the execution process, the adversary can use the 'Set-output', 'Set-state', and 'Get-state' operations to set the output and the state of a party and to learn the state of a party; we refer to such party as *faulty*, and denote by F the set of faulty parties in an execution. In many protocols, one party, say Alice, may *detect* that another party, say Mal, is faulty, typically, by receiving an invalid message from Mal - or simply by *not* receiving a message expected from Mal by a specific 'deadline' (for bounded-delay communication models).

Intuitively, the *No False Accusations (NFA)* requirement predicate π_{NFA} states that a non-faulty entity $a \notin \mathsf{F}$ (Alice), would *never (falsely) accuse* of a fault, another non-faulty entity, $b \notin \mathsf{F}$ (Bob). To properly define this requirement predicate, we first define a convention for one party, say $a \in \mathsf{N}$ (for 'Alice'), to output an Indicator of Accusation, i.e., 'accuse' another party, say $i_{\mathrm{M}} \in \mathsf{N}$ (for 'Mal'), of a fault. Specifically, we say that at event $\hat{e}_A$ of the the execution, entity $ent[\hat{e}_A]$ *accuses* entity i_{M} (Mal), if $out[\hat{e}_A]$ is a triplet of the form $(\mathrm{IA}, i_{\mathrm{M}}, x)$. The last value in this triplet, x, should contain the clock value at the *first* time that Alice accused Mal; we discuss this in section B as the value x is not relevant for the requirement predicate, and is just used as a convenient convention for some protocols.

The No False Accusations (NFA) predicate π_{NFA} checks whether the adversary was able to cause one honest entity, say Alice, to accuse another honest entity, say Bob (i.e., both Alice and Bob are in $\mathsf{N}-\mathsf{F}$). Namely, $\pi_{\mathsf{NFA}}(T)$ returns $\perp$ only if $T.out[e] = (\mathrm{IA}, j, x)$, for some $j \in T.\mathsf{N}$, and both j and $T.ent[e]$ are honest (i.e., $j, T.ent[e] \in T.\mathsf{N} - T.\mathsf{F}$). See Algorithm 31.

Algorithm 31 No False Accusations Predicate $\pi_{\mathsf{NFA}}(T)$

```
1: return ¬(
2:     T.ent[T.e] ∈ T.N − T.F                          ▷ T.ent[T.e] is an honest entity
3:     and ∃j ∈ T.N − T.F, x  s.t. (IA, j, x) ∈ T.out[T.e]   ▷ T.ent[T.e] accused an honest entity
       )
```

As noted above, in an accusation, the output $out[\hat{e}_A]$ contains a triplet of the form $(\mathrm{IA}, i_{\mathrm{M}}, x)$, where x is a clock value and *should* be the clock value at the *first* time that Alice accused Mal. We found this convenient in the definition of protocol-specific requirements where a party may accuse another party multiple times, and the requirement is related to the time of the *first* accuse event. To allow the use of this convention, we define the following 'technical' requirement predicate which merely confirms that honest entities always indicate, in any accuse event, the time of the first time they accused the same entity.

To simplify the predicate, let $fc(i, i_{\mathrm{M}}, T)$ be the value of $T.clk[\hat{e}]$, where $\hat{e}$ is the first event in T in which entity i accused entity $i_{\mathrm{M}} \in T.\mathsf{N}$ (or $\perp$ if no such

event exists). The Use First-Accuse Time (UFAT) predicate π_{UFAT} is defined in Algorithm 32.

Algorithm 32 Use First-Accuse Time Predicate $\pi_{\mathsf{UFAT}}(T)$

return ¬(
  $T.ent[T.e] \in T.\mathsf{N} - T.\mathsf{F}$ ▷ *$T.ent[T.e]$ is an honest entity*
  and $\exists i_{\mathrm{M}} \in T.\mathsf{N}$ **s.t.** $(\mathrm{IA}, i_{\mathrm{M}}, x) \in T.out[T.e]$ **and** $x \neq fc(T.ent[T.e], i_{\mathrm{M}}, T)$ ▷ *$T.ent[T.e]$ did not indicate the first time of accusation in an accusation*
)

B.2.4 Authenticated-sender Communication

The $\pi_{\mathsf{AuthComRcv}}$ model predicate verifies the authentic-sender property for all 'Incoming' events, which means that $out[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ indeed sent this message to $ent[\hat{e}_{\mathrm{R}}]$, during some previous event $\hat{e}_{\mathrm{S}} < \hat{e}_{\mathrm{R}}$. The $\pi_{\mathsf{AuthComRcv}}$ model predicate is shown in Algorithm 33.

Algorithm 33 $\pi_{\mathsf{AuthComRcv}}$ (T) Predicate

return (
  $\forall \hat{e}_{\mathrm{R}} \in \{1, \ldots, T.e\}$:
    if $T.opr[\hat{e}_{\mathrm{R}}] =$ 'Receive': ▷ *For each 'Receive' event*
      and $T.out[\hat{e}_{\mathrm{R}}] \neq \perp$ ▷ *If the 'Receive' event was successful*
      and $T.ent[\hat{e}_{\mathrm{R}}], T.out[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}} \in T.\mathsf{N} - T.\mathsf{F}$ ▷ *And both receiver and sender are honest*
    then $\exists \hat{e}_{\mathrm{S}} < \hat{e}_{\mathrm{R}}$ ▷ *Then there is a previous event*
      s.t. $T.opr[\hat{e}_{\mathrm{S}}] =$ 'Send': ▷ *Which was a 'Send' event*
      and $T.inp[\hat{e}_{\mathrm{S}}].m = T.out[\hat{e}_{\mathrm{R}}].m$ ▷ *Where the input message was $T.out[\hat{e}_R].m$*
      and $T.inp[\hat{e}_{\mathrm{S}}].i_{\mathrm{R}} = T.ent[\hat{e}_{\mathrm{R}}]$ ▷ *And where the intended recipient was $T.ent[\hat{e}_R]$*
      and $T.ent[\hat{e}_{\mathrm{S}}] = T.out[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ ▷ *And where the entity was the sender output in event $\hat{e}_R$*
)

Usually, the base function used with the $\pi_{\mathsf{AuthComRcv}}$ model predicate would be of the form 2^{-l}, where l is the length of some tags or signatures used by the protocol or scheme for authentication. In other words, we would usually allow the adversary to have probability 2^{-l} to cause an entity to receive a forged message.

B.3 Proofs of the Modularity Lemmas

In this section, we give proofs of the modularity lemmas introduced in section 5.

Model Modularity

Lemma 1 (Weaker model satisfaction).

For any set $\mathcal{X}$ of execution-process operations, any (weaker) model $\mathcal{M} = (\pi, \beta)$, and any predicate π', if an adversary $\mathcal{A}$ satisfies the (stronger) model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta)$ (with negligible advantage) then $\mathcal{A}$ satisfies $\mathcal{M}$ (with negligible advantage), namely:

$$\mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \tag{4}$$

Proof. By Def. 2, if $\mathcal{A}$ satisfies model $\widehat{\mathcal{M}}$ with negligible advantage using the set $\mathcal{X}$ of execution-process operations, denoted $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}}$, then:

$$\begin{aligned}&(\forall\ \mathcal{P}, params \in \{0,1\}^*) : \\ &\epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)\end{aligned}$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp\ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

Then, by Def. 1, $\epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$. Now:

$$\begin{aligned}f \equiv &\Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp\ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) = \\ &= \Pr\left[\begin{matrix}(\pi(T) = \perp) \vee (\pi'(T) = \perp)\ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) \\ &\geq \Pr\left[\begin{matrix}\pi(T) = \perp\ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)\end{aligned}$$

Define g as:

$$g \equiv \Pr\left[\begin{matrix}\pi(T) = \perp\ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

Then, by Def. 1, $\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, g\}$. But since, trivially, $g \leq f$, and since $\max\{0, f\}$ is negligible, then $\max\{0, g\}$ must also be negligible, so $\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ is negligible. Therefore:

$$\begin{aligned}&(\forall\ \mathcal{P}, params \in \{0,1\}^*) : \\ &\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)\end{aligned}$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}$, according to Def. 2. □

Lemma 2 (Stronger model satisfaction).

For any set $\mathcal{X}$ of execution-process operations and any two (weaker) models $\mathcal{M} = (\pi, \beta), \mathcal{M}' = (\pi', \beta')$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ (with negligible advantage), then $\mathcal{A}$ satisfies the 'combined' (stronger) model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta + \beta')$ (with negligible advantage), namely:

$$\left(\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \tag{5}$$

Proof. Using Def. 2, the left side implies that for any $\mathcal{P}, params$, we can write:

$$\begin{aligned}&\left(\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})\right)\\ &\wedge \left(\epsilon^{\mathcal{M}'}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})\right)\end{aligned}$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}\pi\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

And define g as:

$$g \equiv \Pr\left[\begin{matrix}\pi'\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta'(params)$$

Then, by Def. 1, $\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$ and $\epsilon^{\mathcal{M}'}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, g\}$. So we have:

$$\begin{aligned}&(\max\{0, f\} \in Negl(params.\mathcal{P}.1^{\kappa}))\\ &\wedge (\max\{0, g\} \in Negl(params.\mathcal{P}.1^{\kappa}))\end{aligned}$$

Which implies that $\max\{0, f + g\}$ must also be negligible. But $\max\{0, f + g\}$ is equivalent to:

$$\begin{aligned}&\Pr\left[\begin{matrix}\pi\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) + \Pr\left[\begin{matrix}\pi'\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta'(params) =\\ &= \Pr\left[\begin{matrix}\pi\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] + \Pr\left[\begin{matrix}\pi'\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))\\ &\geq \Pr\left[\begin{matrix}\left(\pi\left(T\right) = \perp\right) \vee \left(\pi'\left(T\right) = \perp\right) \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params)) =\\ &= \Pr\left[\begin{matrix}\left(\pi \wedge \pi'\right)\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))\end{aligned}$$

Define h as:

$$h = \Pr\left[\begin{matrix}\left(\pi \wedge \pi'\right)\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))$$

Since $h \leq f + g$, and $\max\{0, f + g\}$ is negligible, then it follows that $\max\{0, h\}$ must also be negligible. By Def. 1, $\max\{0, h\} = \epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$. Therefore:

$$(\forall\ \mathcal{P}, params \in \{0,1\}^*) :$$
$$\epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}}$, according to Def. 2. □

Lemma 3 (Requirement satisfaction under stronger model).
If a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ (with negligible advantage) under only $\mathcal{M} = (\pi, \beta)$ using the execution-process operations set $\mathcal{X}$, then for any predicate π', $\mathcal{P}$ ensures $\mathcal{R}$ (with negligible advantage) under the 'combined' model $\widehat{\mathcal{M}} \equiv (\pi \wedge \pi', \beta)$, using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{poly} \mathcal{R} \tag{6}$$

Proof. By Def. 3, the left side implies that we can write:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\Big) :$$
$$\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)$$

Let $\mathcal{M}' = (\pi', \beta)$. By the first part of the Model Modularity lemma, $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}} \Rightarrow \Big(\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}'\Big)$. Therefore, we have:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \widehat{\mathcal{M}}\Big) :$$
$$\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)$$

Which is the definition of $\mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{\text{poly}} \mathcal{R}$, according to Def. 3. □

Lemma 4 (Model monotonicity). *Let $\mathcal{X}$ be any set of execution-process operations, π be a predicate, and β, β' be two base functions. Define the function $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all params. Let $\mathcal{M} = (\pi, \beta)$ and $\widehat{\mathcal{M}} = (\pi, \hat{\beta})$. Then holds:*

$$(\forall \mathcal{A}) \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \tag{8}$$

Proof. By Def. 2, if $\mathcal{A}$ satisfies model $\mathcal{M}$ with negligible advantage using the set $\mathcal{X}$ of execution-process operations, denoted $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}$, then:

$$(\forall\ \mathcal{P}, params \in \{0,1\}^*) :$$
$$\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^\kappa)$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

And define g as:

$$g \equiv \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \hat{\beta}(params)$$

Then $f \geq g$, because $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all $params$. Notice that by Def. 1, $\epsilon^{\mathcal{M}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$ and $\max\{0, g\} = \epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$.

Since $g \leq f$, and $\max\{0, f\}$ is negligible, then $\max\{0, g\} = \epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ must also be negligible. This means that:

$$(\forall\ \mathcal{P}, params \in \{0,1\}^*) :$$
$$\epsilon^{\widehat{\mathcal{M}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}}$, according to Def. 2. □

Requirement Modularity

Lemma 5 (Weaker requirement satisfaction).
For any set $\mathcal{X}$ of execution-process operations, any model $\mathcal{M}$, any (weaker) requirement $\mathcal{R} = (\pi, \beta)$, and any predicate π', if a protocol $\mathcal{P}$ ensures the (stronger) requirement $\widehat{\mathcal{R}} \equiv (\pi \wedge \pi', \beta)$ (with negligible advantage) under model $\mathcal{M}$, then $\mathcal{P}$ ensures $\mathcal{R}$ (with negligible advantage) under model $\mathcal{M}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R} \tag{9}$$

Proof. By Def. 3, the left side implies that:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\Big) :$$
$$\epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

And define g as:

$$g \equiv \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

Then, by Def. 1, $\epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$ and $\max\{0, g\} = \epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$. Notice that:

$$\begin{aligned}
&\Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) = \\
&= \Pr\left[\begin{matrix}(\pi(T) = \perp) \vee (\pi'(T) = \perp) \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) \\
&\geq \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)
\end{aligned}$$

So $f \geq g$. Since $\max\{0, f\}$ is negligible and $g \leq f$, then $\max\{0, g\} = \epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ must also be negligible. Therefore:

$$\begin{aligned}
&\left(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\right) : \\
&\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})
\end{aligned}$$

Which is the definition of $\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\text{poly}} \mathcal{R}$, according to Def. 3.

Lemma 6 (Stronger requirement satisfaction).

For any set $\mathcal{X}$ of execution-process operations, any model $\mathcal{M}$, and any two (weaker) requirements $\mathcal{R} = (\pi, \beta), \mathcal{R}' = (\pi', \beta')$, if a protocol $\mathcal{P}$ ensures both $\mathcal{R}$ and $\mathcal{R}'$ (with negligible advantage) under model $\mathcal{M}$, then $\mathcal{P}$ ensures the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv (\pi \wedge \pi', \beta + \beta')$ (with negligible advantage) under model $\mathcal{M}$, namely:

$$\left(\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \widehat{\mathcal{R}} \tag{10}$$

Proof. Using Def. 3, the left side implies that:

$$\begin{aligned}
&\left(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\right) : \\
&\left(\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})\right) \\
&\wedge \left(\epsilon^{\mathcal{R}'}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})\right)
\end{aligned}$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

And define g as:

$$g \equiv \Pr\left[\begin{matrix}\pi'(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta'(params)$$

Then, by Def. 1, $\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$ and $\epsilon^{\mathcal{R}'}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, g\}$. Since $\max\{0, f\}$ is negligible and $\max\{0, g\}$ is negligible, then $\max\{0, f+g\}$ must also be negligible. Notice that $f + g$ is equivalent to:

$$\Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params) + \Pr\left[\begin{matrix}\pi'(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta'(params) =$$
$$= \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] + \Pr\left[\begin{matrix}\pi'(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))$$
$$\geq \Pr\left[\begin{matrix}(\pi(T) = \perp) \vee (\pi'(T) = \perp) \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params)) =$$
$$= \Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))$$

Define h as:

$$h \equiv \Pr\left[\begin{matrix}(\pi \wedge \pi')(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - (\beta(params) + \beta'(params))$$

Then $f + g \geq h$. Since $h \leq f + g$ and $\max\{0, f + g\}$ is negligible, then $\max\{0, h\}$ must also be negligible. By Def. 1, $\max\{0, h\} = \epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ Therefore:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\Big) :$$
$$\epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})$$

Which is the definition of $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\text{poly}} \widehat{\mathcal{R}}$, according to Def. 3. □

Lemma 7 (Requirement monotonicity). *Let $\mathcal{X}$ be any set of execution-process operations, $\mathcal{M}$ be a model, π be a predicate, and β, β' be two base functions. Define the function $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all params. Let $\mathcal{R} = (\pi, \beta)$ and $\widehat{\mathcal{R}} = (\pi, \hat{\beta})$. Then holds:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R} \Rightarrow \mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \widehat{\mathcal{R}} \tag{11}$$

Proof. By Def. 3, if $\mathcal{P}$ satisfies requirement $\mathcal{R}$ with negligible advantage under model $\mathcal{M}$ using the set $\mathcal{X}$ of execution-process operations, denoted $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\text{poly}} \mathcal{R}$, then:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models_{\text{poly}} \mathcal{M}\Big) :$$
$$\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})$$

Define f as:

$$f \equiv \Pr\left[\begin{matrix}\pi(T) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{matrix}\right] - \beta(params)$$

And define g as:

$$g \equiv \Pr\left[\begin{array}{l}\pi\left(T\right) = \perp \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)\end{array}\right] - \hat{\beta}(params)$$

Then $f \geq g$, because $\hat{\beta}(params) = \max\{\beta(params), \beta'(params)\}$ for all $params$. Notice that by Def. 1, $\epsilon^{\mathcal{R}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) = \max\{0, f\}$ and $\max\{0, g\} = \epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$.

Since $g \leq f$ and $\max\{0, f\}$ is negligible, then $\max\{0, g\} = \epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params)$ must also be negligible. This means that:

$$\Big(\forall\ \mathcal{A} \in PPT, params \in \{0,1\}^* \mid \mathcal{A} \models^{}_{\text{poly}} \mathcal{M}\Big) :$$
$$\epsilon^{\widehat{\mathcal{R}}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \in Negl(params.\mathcal{P}.1^{\kappa})$$

Which is the definition of $\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\text{poly}} \widehat{\mathcal{R}}$, according to Def. 3. □