MoSS: Modular Security Specifications 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. Applied cryptographic protocols have to meet a rich set of security requirements under diverse environments and against diverse adversaries. However, currently used security specifications, based on either simulation [10,24] (e.g., 'ideal functionality' in UC) or games [7,26], are *monolithic*, combining together different aspects of protocol requirements, environment and assumptions. Such specifications are complex, error-prone, and foil reusability, modular analysis and incremental design.

We present the *Modular Security Specifications (MoSS) framework*, which cleanly separates each *security requirement* (goal) which a protocol should achieve, from the environment and *model* (assumptions) under which the requirement should be ensured. This modularity allows us to reuse individual models and requirements across different protocols and tasks, and to compare protocols for the same task, either under different assumptions (models) or satisfying different sets of requirements. MoSS is flexible and extendable, e.g., it can support both *asymptotic* and *concrete* definitions for security.

We demonstrate the applicability of MoSS to two applications: secure broadcast protocols and PKI schemes.

1 Introduction

Precise and correct models, requirements and proofs are the best way to ensure security. Unfortunately, it is hard to write them and easy-to-make subtle errors often result in vulnerabilities and exploits; this happens even to the best cryptographers, with the notable exception of the reader. Furthermore, 'the devil is in the details'; minor details of the models and requirements can be very significant, and any inaccuracies or small changes may invalidate proofs.

Provable security has its roots in the seminal works rigorously proving security for constructions of cryptographic primitives such as signature schemes [17], encryption schemes [16] and pseudorandom functions [15]. Nowadays, provable security under well-defined assumptions is expected from any work presenting a new design or a new cryptographic primitive. With time, the expectation of a provably-secure design has extended to applied cryptographic protocols, with

[⋆] To be presented in Crypto'2021

seminal works such as [4,6]. After repeated discoveries of serious vulnerabilities in 'intuitively-designed' protocols [14], proofs of security are expected, necessary and appreciated by practitioners. However, provable security is notoriously challenging and error-prone for applied cryptographic protocols, which often aim to achieve complex goals under diverse assumptions intended to reflect real-world deployment scenarios.

MoSS: A Modular Approach to Specifications. For proofs of security to be meaningful in practice, protocols should be defined and proven with respect to a particular choice of well-defined, realistic *models* (assumptions) and *requirements* (properties/goals), and for a specific *execution process*. In MoSS, a *specification* consists of *individually-defined* models and requirements given a specific execution process, which is also defined *independently* of models and requirements and can be extended if needed. By fundamentally decoupling these components, they can be easily updated, replaced, reused or removed as needed. This level of modularity is particularly beneficial for applied protocols due to the high number of requirements and models such protocols may need to consider (see Figure 1) during design and analysis.

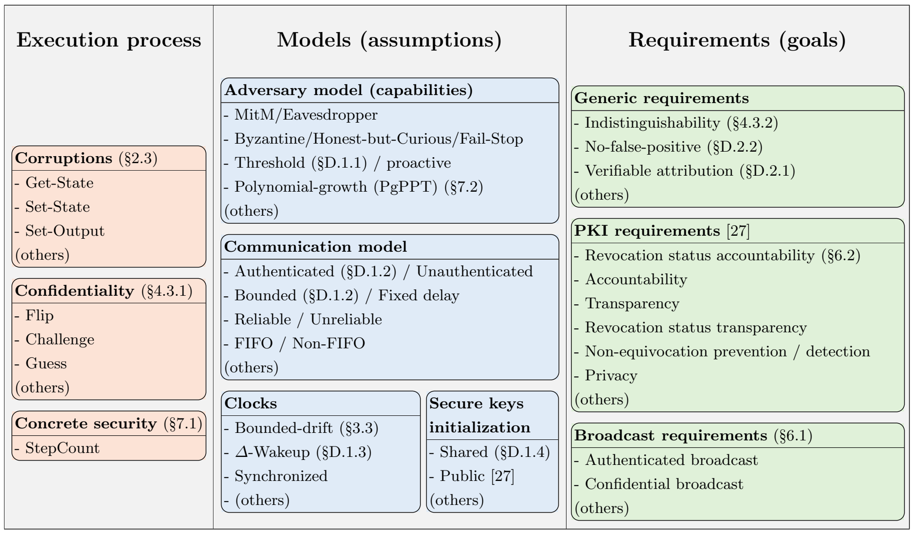

Fig. 1: The MoSS framework allows security to be specified *modularly*, i.e., 'à la carte', with respect to a set of individually-defined models (assumptions), requirements (properties/goals) and even operations of the execution process. Models, requirements and operations defined in this paper or in [27] are marked accordingly. Many models, and some ('generic') requirements, are applicable to different types of protocols.

Models are used to reflect different assumptions made for a protocol, such as the adversary capabilities, communication (e.g., delays and reliability), synchronization, initialization and more. For each 'category' of assumptions, there are multiple options available: e.g., MitM or eavesdropper for the adversary model; threshold for the corruption model; asynchronous, synchronous, or bounded delay for the communication delays model; or asynchronous, synchronous, syntonized, bounded drift for the clock synchronization model. In practice, models may be shared across protocols with different goals (e.g., many protocols use a synchronous communication model regardless of their specific goals). At the same time, protocols with the same goals may use different models (e.g., either synchronous or asynchronous).

Requirements refer to properties or goals a protocol aims for. As before, protocols for the same problem may achieve different requirements, which may be comparable (e.g., equivocation detection vs. equivocation prevention) or not (e.g., accountability vs. transparency). While many requirements are task-specific, some *generic* requirements are applicable across different tasks; e.g., a *no false positive* requirement to ensure that an honest entity should never be considered 'malicious' by another honest entity.

Execution process. MoSS executions are defined by a well-defined execution process (see Algorithm 1) which takes as input a protocol to execute, an adversary and parameters. The MoSS execution process can be customized using *execution operations.* We define such operations to extend the 'core' execution process with operations allowing confidentiality (indistinguishability) definitions, entity-corruptions, and concrete security.

Related work. Currently, there are two main approaches for defining security specifications: simulation-based and game-based.

The *simulation-based approach*, most notably Universal Composability (UC) [10], defines security as indistinguishability between executions of a given protocol and executions of an 'ideal functionality', which blends together the model (assumptions) and requirements. There are multiple extensions to UC such as iUC, GNUC, IITM and simplified-UC [9, 11, 18, 20, 28], and other simulation-based frameworks such as constructive cryptography (CC) [23, 24] and reactive systems [1]. Each of these variants defines a specific but fixed execution model. The popularity of the simulation-based approach is largely due to its support for *secure composition* of protocols; however, arguably, this approach results in more complex specifications and proofs, which may be more difficult to understand and verify, especially for practitioners.

The *game-based approach* [7,26] is widely-adopted and more successful among practitioners, due to its simpler, more intuitive definitions and proofs of security. In this approach, for each task, a particular *game* is defined, which combines a specific model, one or more requirements and a specific execution process.

Our motivation for this work stems from an observation that the standard practice in provable security is to define and analyze the security of schemes and protocols using *monolithic security specifications* (an ideal functionality[4] or a game), in effect combining security requirements with different aspects of the model and the execution process. Even though requirements and models are of-

[4] Simulation-based specifications even combine all requirements into the *same* ideal functionality.

	Specifications			Multiple specifications	Composability
	Exec Process	Models	Requirements		
MoSS	Modular	Modular	Modular	Yes	?
Simulation-based (e.g., UC [10])	Fixed	Combined		No	Yes
Game-based [7, 26]	Combined			Yes	No

Table 1: A comparison of specifications produced by different provable-security frameworks, where an execution process is used to define an execution of a protocol, models reflect the assumptions under which the protocol requirements must hold. Multiple specifications allow to independently define different protocol goals. Composability allows to arbitrarily compose different protocols and retain security.

ten individually presented in their informal descriptions, the designers and readers have to validate directly that the formal, monolithic specifications correctly reflect the informally-discussed models and assumptions. Such an approach is especially challenging for applied protocols with complex requirements and models, and it stands in sharp contrast to the standard engineering approach, where specifications are gradually developed and carefully verified at each step, often using automated tools. While there exist powerful tools to validate security of cryptographic protocols [2], there are no such tools to validate the *specifications*.

In Table 1, we compare MoSS to game-based specifications and simulation-based specifications.

Advantages of MoSS. The main advantage of MoSS is its complete *modularity*; a security specification is defined as a specific collection of requirements, models and execution process operations, and the same protocol can be proven to satisfy multiple specifications.

MoSS specifications can be *reused* across different protocols. For example, in Appendix B we present a simplified instance of an authenticated-broadcast protocol assuming a (well-defined) bounded delay and bounded clock-drift models. Appendix D includes more models and requirements. These models and requirements are *generic* and can be reused for different problems.

The use of separate, focused models and requirements also allows a *gradual protocol development and analysis*. To illustrate, we first analyze the authenticated-broadcast protocol assuming only a secure shared-key initialization model, which suffices to ensure authenticity but not freshness. We then show that the protocol also achieves freshness when we also assume a bounded clock-drift model. Lastly, we show that by additionally assuming a bounded-delay communication model, we can ensure a bounded delay for the broadcast protocol. This gradual approach makes the analysis easier to perform and understand (and to identify any design flaws early on), especially when compared to proving such properties using monolithic specifications (all at once). Using MoSS is a bit like playing Lego with models and requirements!

Concrete security [5] is especially important for protocols used in practice as it allows to more precisely define security of a given protocol and to properly select security parameters, in contrast to asymptotic security. Unlike UC, which only provides partial support for concrete security[5], due to its modularity, MoSS supports concrete security in a way we consider simple, elegant and precise (see Section 7.1.1).

Reactive polynomial time. As pointed out in [10, 19], the 'classical' notion of PPT algorithms is not sufficient for analysis of reactive systems, where the same protocol (and adversary) can be invoked many times. This issue is addressed by later versions of UC and in some other recent frameworks, e.g., GNUC [18], but mostly ignored by game-based definitions. This flexibility of MoSS allows it to handle these aspects relatively simply (see Section 7.2 and Appendix C).

Modularity lemmas. In Section 5, we present several *asymptotic-security modularity lemmas* allowing to combine 'simple' models and requirements into composite models and requirements to fully take advantage of MoSS's modularity. We provide proofs and corresponding concrete-security modularity lemmas in Appendices E and F.

Limitations of MoSS. Currently, MoSS has two significant limitations: the lack of *computer-aided tools*, available for both game-based and simulation-based approaches [3,8,25], and the lack of *composability*, an important property proven for most simulation-based frameworks, most notably UC [10].

We believe that MoSS is amenable to computer-aided tools. For example, a tool may transform the modular MoSS specifications into a monolithic game or an ideal functionality, allowing to use the existing computer-aided tools. However, development of such tools is clearly a challenge yet to be met. Another open challenge is to prove a composability property directly for MoSS specifications.

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.

Real-world application of MoSS: PKI. Public Key Infrastructure (PKI) schemes, a critical component of applied cryptography, amply illustrate the challenges of applying provable security in practice and serve as a good example of how MoSS might benefit practical protocols. Current PKI systems are mostly based on the X.509 standard [13], but there are many other proposals, most notably, Certificate Transparency (CT) [21], which adds significant goals and cryptographic mechanisms. 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, [27] presented the first rigorous study of practical[6] 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 [27] reuses our bounded-delay and bounded-drift models; similarly, follow-up work

[5] UC does not support concrete bounds on the adversary or environment runtime. Even this limited support was added in more recent versions.

[6] Grossly-simplified PKI ideal functionalities were studied, e.g., in [18], but without considering even basic aspects such as revocation and expiration.

is expected to use the models and requirement predicates defined in [27] to prove security for additional PKI schemes, e.g., Certificate Transparency, being standardized by the IETF [21].

Organization. Section 2 introduces **Exec**, the adversary-driven execution process. Section 3 and Section 4 present the model and requirement specifications, respectively, and App. D describes several additional examples of useful specifications. Section 5 presents the modularity lemmas. Section 6 shows how to apply MoSS to two different applications, a simplified authenticated broadcast protocol, further described in App. B, and PKI schemes. Section 7 describes two important extensions of the framework to achieve concrete security and to ensure polynomial runtime, with additional details in App. C. We conclude and discuss future work in Section 8.

2 Execution Process

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. In this section, we present this execution process, which defines the execution of a given protocol $\mathcal{P}$ 'controlled' by a given adversary $\mathcal{A}$. We say that it is 'adversary-driven' since the adversary controls all inputs and invocations of the entities running the protocol.

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 Fig. 2 (see also more elaborate illustration in Fig. 3 in App. A), 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 consist 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. 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

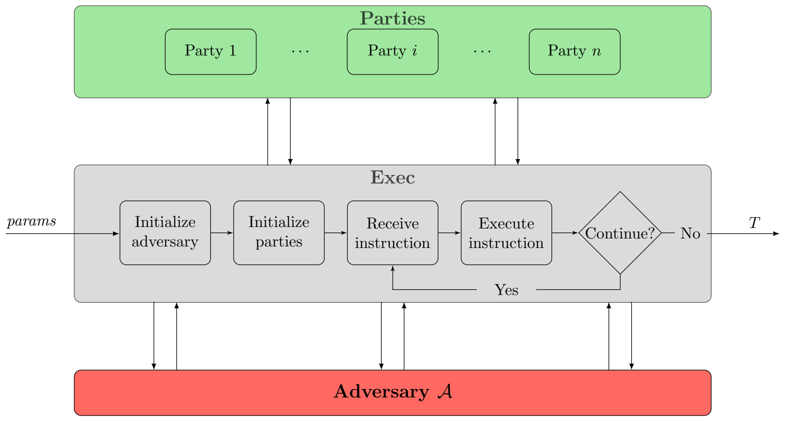

Fig. 2: A high level overview of the MoSS's execution process showing the interactions between the parties to the protocol and the adversary in $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$.

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 an 'algorithm' (in PPT) although we do not consider the operation as part of the input, i.e., formally, $\mathcal{P}$ maps from the operations (given as strings) to algorithms; this can be interpreted as $\mathcal{P}$ accepting the 'label' as additional input and calling the appropriate 'subroutine', making it essentially a single PPT algorithm.

Algorithm 1 uses the standard *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. Even the adversary does not control e, although, the adversary can keep track of it in its state, since it is invoked (twice) in every round. 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.

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 $params.\mathcal{P}$.

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:     $e \leftarrow e + 1$    ▷ Advance the loop counter
6:     $(ent[e], opr[e], inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$    ▷ $\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event $e$
7:     $(s_{ent[e]}, out[e]) \leftarrow \mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$
8:     $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}\,(s_{\mathcal{A}}, out[e])$    ▷ $\mathcal{A}$ 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
```

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 a complex initialization process can use other operations defined in $\mathcal{P}$ in the main execution loop (lines 4-9), to implement an initialization process which cannot be performed via a 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 The Extendable Execution Process

In Section 2.1, we described the design of the generic $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$ execution process, which imposes only some basic limitations. We now describe the *extendable* execution process $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$, an extension of $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$, which provides additional flexibility with only few changes to $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$. The extendable execution process $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ allows MoSS to (1) handle different kinds of entity-corruptions (described next), (2) define indistinguishability requirements (Section 4.3) and (3) support concrete security (Section C); other applications may be found.

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[7] $\mathcal{X}$ 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 functionality 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 Section 2.3.

Changes to the $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$ execution process. The input parameters consist 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 decide 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.

[7] We use the term 'set', but note that $\mathcal{X}$ is defined as a single PPT algorithm, similarly to how $\mathcal{P}$ is defined.

Algorithm 2 Extendible 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.A, params.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: $\boldsymbol{s_{\mathcal{X}} \leftarrow \mathcal{X}[\text{'Init'}](params)}$    ▷ Initial exec state
4: $e \leftarrow 0$    ▷ Initialize loop's counter
5: repeat
6:     $e \leftarrow e + 1$    ▷ Advance the loop counter
7:     $(ent[e], opr[e], \mathbf{type[e]}, inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$    ▷ A selects entity ent[e], operation opr[e], input inp[e], clock clk[e], and real time τ[e] for event e
8:     if $\boldsymbol{type[e] = \text{'}\mathcal{X}\text{'}}$ then    ▷ If A chose to invoke an operation from X.
9:         $\boldsymbol{(s_{\mathcal{X}}, s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{X}[opr[e]](s_{\mathcal{X}}, s_{ent[e]}, inp[e], clk[e], ent[e])}$
10:    else    ▷ A chose to invoke an operation from P.
11:        $(s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}[opr[e]](s_{ent[e]}, inp[e], clk[e])$
12:    end if
13:    $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}(s_{\mathcal{A}}, out[e])$    ▷ A decides when to terminate the loop ($out_{\mathcal{A}} \neq \perp$), based on out[e]
14: until $out_{\mathcal{A}} \neq \perp$
15: $T \leftarrow (out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], type[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], sec\text{-}out[\cdot][\cdot])$
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 one 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 $\mathcal{X}$). (If $type[e] \notin \{\text{'}\mathcal{X}\text{'}, \text{'}\mathcal{P}\text{'}\}$, then the execution process assumes that the operation is protocol-specific.) Afterwards, the event is executed (lines 8-11) through the appropriate algorithm, based on the operation type, either $\mathcal{X}$, if $type[e] = \text{'}\mathcal{X}\text{'}$, or $\mathcal{P}$ otherwise.

The termination phase (lines 15-16) is the same as before, but also includes in the transcript the $type[\cdot]$ values and the $sec\text{-}out[\cdot][\cdot]$ for all invoked events. 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.

Note: First, any set of execution operations $\mathcal{X}$ is assumed to contain an 'Init' operation, and we may omit the 'Init' operation from the notation when specifying $\mathcal{X}$; if it is omitted, the 'default' 'Init' operation is assumed, which simply outputs *params*. Second, we assume that $\mathcal{X}$ operations are always defined such that whenever $\mathcal{X}$ is invoked, it does not run $\mathcal{A}$ and only runs $\mathcal{P}$ at most once (per invocation of $\mathcal{X}$). Third, in lines 7 and 13, the operation to $\mathcal{A}$ is not explicitly written in the pseudocode. We assume that in fact nothing is given to $\mathcal{A}$ for the operation (length 0) - this implies that $\mathcal{A}$ will not be re-initialized during the execution process.

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 App. B.2. We define the 'Sec-in' operation in Equation 1.[8]

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

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*

[8] We use $\equiv$ to mean 'is defined as'.

(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).

See discussion in App. D.1.1, and an example: use of these 'fault operations' to define the *threshold security* model $\mathcal{M}^{|\mathsf{F}|\leq f}$, assumed by many protocols.

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 model 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 Models

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 two important concepts of MoSS: a *model* $\mathcal{M}$, used to define assumptions about the adversary and the execution, and *specifications* (π, β). We use specifications to define both models (in this section) and requirements (in section 4).

A MoSS (model/requirement) specification is a pair of functions (π, β), where $\pi(T, params) \in \{\top, \bot\}$ is called the *predicate* and $\beta(params) \in [0, 1]$ is the *base (probability) function*. The predicate π is applied to the execution-transcript T and defines whether the adversary 'won' or 'lost'. The base function β is the 'inherent' probability of the adversary 'winning'; it is often simply zero ($\beta(x) = 0$), e.g., for forgery in a signature scheme, but sometimes a constant such as half (for indistinguishability specifications) or a function such as 2^{-l} (e.g., for l bit MAC) of the parameters *params*.

MoSS models are defined as a set of (one or more) specifications, i.e., $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$. When the model contains only one specification, we may abuse notation and write $\mathcal{M} = (\pi, \beta)$ for convenience.

For example, consider a model $\mathcal{M} = (\pi, 0)$. Intuitively, adversary $\mathcal{A}$*satisfies model* $(\pi, 0)$, if for (almost) all execution-transcripts T of $\mathcal{A}$, predicate π holds, i.e.: $\pi(T, params) = $ 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 that the adversary holds over the execution is used 'with great responsibility'.*

The separation between the execution process and the model allows us to use the same - relatively simple - execution process, for the analysis of many different protocols, under different models (of adversary, environment and capabilities). Furthermore, it allows us to define multiple simple models, each focusing on a different assumption or restriction, and require that the adversary satisfy all of them.

As depicted in Figure 1, the model captures all of the assumptions regarding the environment and the capabilities of the adversary, including aspects typically covered by the (often informal) *communication model, synchronization model* and *adversary model*:

Adversary model: The adversary capabilities such as MitM vs. eavesdropper, entity corruption capabilities (e.g., threshold or proactive security), computational capabilities and more.

Communication model: The properties of the underlying communication mechanism, such as reliable or unreliable communication, FIFO or non-FIFO, authenticated or not, bounded delay, fixed delay or asynchronous, and so on.

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

The definitions of models and their predicates are often simple to write and understand - and yet, reusable across works.

In Sec. 3.1, we define the concept of a specification. 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. Additional examples of models are given later in this paper, mainly in D.1.

3.1 Specifications

We next define the *specification*, used to define both *models* and *requirements*.

A specification is a pair (π, β), where π is the *specification predicate* and β is the *base function*. A *specification predicate* is a predicate whose inputs are 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 integrity specifications, e.g. forgery, the base function is often either zero or 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 predicate π using execution operations $\mathcal{X}$, as a function of the parameters *params*. This the probability that $\pi(T, params) = \bot$, 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 predicate π using execution operations $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X} \in PPT$ and let π be a specification predicate. The advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π using execution operations $\mathcal{X}$ is defined as:*

$$\epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \overset{def}{=} \Pr\begin{bmatrix} \pi\left(T, params\right) = \bot, \ \textit{where} \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params) \end{bmatrix} \tag{2}$$

3.2 Model-Satisfying Adversary

Models are sets of specifications, used to restrict the capabilities of the adversary and the events in the execution process. This includes limiting of the possible faults, defining initialization assumptions, and defining the communication and synchronization models. We check whether a given adversary $\mathcal{A}$ followed the restrictions of a given model $\mathcal{M}$ in a given execution by examining whether a random transcript T of the execution satisfies each of the model's specification predicates. Next, we define what it means for adversary $\mathcal{A}$ to *satisfy model $\mathcal{M}$ with negligible advantage using execution operations $\mathcal{X}$*.

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 $\mathcal{M}$ be a set of specifications, i.e., $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$. We say that adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with negligible advantage using execution operations $\mathcal{X}$, denoted $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}$, if for every protocol $\mathcal{P} \in PPT$, $params \in \{0,1\}^*$, and specification $(\pi, \beta) \in \mathcal{M}$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π using $\mathcal{X}$ is at most negligibly greater than $\beta(params)$, i.e.:*

$$\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \overset{def}{=} \begin{bmatrix} \forall \ \left(\mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}\right) : \\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl\left(|params|\right) \end{bmatrix} \tag{3}$$

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

To demonstrate a definition of a model predicate, we present the $\mathcal{M}^{\mathrm{Drift}}_{\Delta_{clk}}$ model, defined as $\mathcal{M}^{\mathrm{Drift}}_{\Delta_{clk}} = (\pi^{\mathrm{Drift}}_{\Delta_{clk}}, \beta^{\mathrm{Drift}}_{\Delta_{clk}})$, where $\beta^{\mathrm{Drift}}_{\Delta_{clk}} = 0$ (i.e., the base function is always zero, like for most integrity properties). The predicate $\pi^{\mathrm{Drift}}_{\Delta_{clk}}$ 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

$\varDelta_{clk} = 0$, 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 $\pi^{\text{Drift}}_{\varDelta_{clk}}$ (T, params) Predicate

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$:                          ▷ For each event
3:         $|T.clk[\hat{e}] - T.\tau[\hat{e}]| \leq \varDelta_{clk}$       ▷ Local clock is within $\varDelta_{clk}$ drift from real time
4:         and if $\hat{e} \geq 2$ then $T.\tau[\hat{e}] \geq T.\tau[\hat{e}-1]$   ▷ In each consecutive event, the real time difference is monotonically increasing
   )
```

4 Requirements

In this section we define and discuss *requirements*. Like a model, a *requirement* is a set of specifications $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$. When the requirement contains only one specification, we may abuse notation and write $\mathcal{R} = (\pi, \beta)$ for convenience. Each requirement specification $(\pi, \beta) \in \mathcal{R}$ includes a predicate (π) and a base function (β). A requirement defines one or more properties that a protocol aims to achieve, e.g., security, correctness or liveness requirements. By separating between models and requirements, MoSS obtains modularity and reuse; different protocols may satisfy the same requirements but use different models, and the same models can be reused for different protocols, designed to satisfy different requirements.

The separation between the definition of the model and of the requirements also allows definition of *generic requirement predicates.*, which are applicable to protocols designed for different tasks, which share some basic goals. We identify several 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 D.2).

4.1 Model-Secure Requirements

We next define what it means for a protocol to satisfy a requirement under some model. First, consider a requirement $\mathcal{R} = (\pi, \beta)$, which contains just one specification, and let b be the outcome of π applied to (T, params), where T is a transcript of the execution process ($T = \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(\mathit{params})$) and params are the parameters, i.e., $b \leftarrow \pi(T, \mathit{params})$; if $b = \bot$ 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.*

We now 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 $\mathcal{R}$ be a set of specifications, i.e., $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$. 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 using execution operations $\mathcal{X}$ and for every parameters $params \in \{0,1\}^*$, where $params = (params.\mathcal{A}, params.\mathcal{P})$ and $|params.\mathcal{P}| \geq |params.\mathcal{A}|$*[9]*, for every specification $(\pi, \beta) \in \mathcal{R}$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π using $\mathcal{X}$ is at most negligibly greater than $\beta(params)$, i.e:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R} \overset{def}{=} \begin{bmatrix} (\forall\ \mathcal{A}\ s.t.\ \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \mathcal{R}) : \\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl\left(|params|\right) \end{bmatrix} \tag{4}$$

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

As an example of a requirement, 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 $\pi_{\mathsf{AuthComRcv}_{\Delta}}(T, params)$ Predicate

```
1: return (
2:     $\forall \hat{e}_R \in \{1, \ldots, T.e\}$:
3:         if $T.out[\hat{e}_R] = (\text{'Receive'}, m)$:        ▷ For each event where a message is received
4:         $\exists \hat{e}_B \in \{1, \ldots, \hat{e}_R - 1\}$       ▷ There is a previous event
5:             s.t. $T.opr[\hat{e}_B] = \text{'Broadcast'}$:      ▷ Which was a 'Broadcast' event
6:             and $T.\tau[\hat{e}_R] - T.\tau[\hat{e}_B] \leq \Delta$   ▷ Within the last $\Delta$ real time
7:             and $T.inp[\hat{e}_B].m = m$                      ▷ Where the input message was $m$
   )
```

4.3 Supporting Confidentiality and Indistinguishability

The MoSS framework supports specifications for diverse goals and scenarios. We 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'.

[9] We require $|params.\mathcal{P}| \geq |params.\mathcal{A}|$ to prevent a run where $params.\mathcal{A}$ is absurdly longer than $params.\mathcal{P}$, allowing $\mathcal{A}$ to run time exponential in $params.\mathcal{P}$.

- 'Flip': selects a uniformly random bit $s_{\mathcal{X}}.b$ via coin flip, i.e., $s_{\mathcal{X}}.b \overset{R}{\leftarrow} \{0,1\}$.
- '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 $sec\text{-}out[e]$. The result is put in *sec-out* to prevent the adversary from accessing 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 $IND^{\pi_{\mathsf{MsgConf}}}$: Message Confidentiality for Encrypted Communication

To illustrate how the aforementioned operations can be used in practice, we define the indistinguishability requirement predicate IND^{π} in Algorithm 5. IND^{π} 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 IND^{π} predicate returns the *negation* of whether the adversary guessed correctly during the last event of the execution.

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

```
1: return ¬(
2:     $T.type[T.e] = \text{'}\mathcal{X}\text{'}$
3:     and $T.opr[T.e]$ = 'Guess' and $T.sec\text{-}out[T.e] = \top$    ▷ The last event is a 'Guess' event and $\mathcal{A}$ guessed correctly
4:     and $\pi(T, params)$    ▷ The model predicate $\pi$ was met
    )
```

We can use IND^{π} to define more specific predicates; for example, we use the π_{MsgConf} predicate (Algorithm 6) to define $IND^{\pi_{\mathsf{MsgConf}}}$, 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 (2) a 'Receive' operation, which takes as input an encrypted message and decrypts it.

The π_{MsgConf} specification 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 π_{MsgConf} (T, params) Predicate

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$ s.t.
           $T.type[\hat{e}] = `\mathcal{X}'$ and $T.opr[\hat{e}] =$ 'Challenge':        ▷ Every 'Challenge' event
3:         $T.inp[\hat{e}].opr =$ 'Send'                                          ▷ was for 'Send' operations only
4:         and $|T.inp[\hat{e}].inp[0].m| = |T.inp[\hat{e}].inp[1].m|$            ▷ with equal length messages
5:             and $\exists\, i_{\mathrm{S}}, i_{\mathrm{R}} \in T.\mathsf{N}$ s.t.   ▷ There is one specific sender $i_S$ and one specific receiver $i_R$
6:             $T.inp[\hat{e}].inp[0].i_{\mathrm{R}} = T.inp[\hat{e}].inp[1].i_{\mathrm{R}} = i_{\mathrm{R}}$   ▷ $i_R$ is the recipient for both messages
7:             and $T.ent[\hat{e}] = i_{\mathrm{S}}$                              ▷ And $i_S$ is the sender
8:             and $\nexists\, \hat{e}'$ s.t. $T.opr[\hat{e}'] =$ 'Receive'       ▷ And there is no event $\hat{e}'$
9:             and $T.inp[\hat{e}'].c = T.out[\hat{e}].c$
               and $T.ent[\hat{e}'] = i_{\mathrm{R}}$                             ▷ Where $\mathcal{A}$ uses the 'Receive' event to decrypt the output of the challenge
               and $T.inp[\hat{e}'].i_{\mathrm{S}} = i_{\mathrm{S}}$
   )
```

5 Modularity Lemmas

MoSS models and requirements are defined as sets of specifications, so they can easily be combined by simply taking the union of sets. There are some intuitive properties one expects to hold for such modular combinations of models or requirements. In this section we present the model and requirement *modularity* lemmas, which essentially formalize these intuitive properties. The lemmas can be used in analysis of applied protocols, e.g., to allow a proof of a requirement under a weak model to be used as part of a proof of a more complex requirement which holds only under a stronger model. We believe that they may be helpful when applying formal methods, e.g., for automated verification and generation of proofs.

In this section, we present the asymptotic-security lemmas; the (straightforward) proofs of the asymptotic-security lemmas are in App. E. The concrete-security lemmas and their proofs are in App. F.

In the following lemmas, we describe model $\widehat{\mathcal{M}}$ as *stronger* than a model $\mathcal{M}$ (and $\mathcal{M}$ is *weaker* than $\widehat{\mathcal{M}}$) if $\widehat{\mathcal{M}}$ includes all the specifications of $\mathcal{M}$, i.e., $\mathcal{M} \subseteq \widehat{\mathcal{M}}$. 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 $\widehat{\mathcal{R}}$ includes all the specifications of $\mathcal{R}$, i.e., $\mathcal{R} \subseteq \widehat{\mathcal{R}}$. Basically, stronger models enforce more (or equal) constraints on the adversary or other assumptions, compared to weaker ones, while stronger requirements represent more (or equal) properties achieved by a protocol or scheme, compared to weaker ones.

5.1 Asymptotic-Security Model Modularity Lemmas

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 lemma is *Model Monotonicity Lemma (asymptotic-security)*. It shows that if an adversary $\mathcal{A}$ satisfies a stronger model $\widehat{\mathcal{M}}$, then $\mathcal{A}$ also satisfies any model that is weaker than $\widehat{\mathcal{M}}$.

Lemma 1 (Model Monotonicity Lemma (asymptotic-security)).
For any set $\mathcal{X}$ of execution process operations, for any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if an adversary $\mathcal{A}$ satisfies $\widehat{\mathcal{M}}$ with negligible advantage using $\mathcal{X}$ then $\mathcal{A}$ satisfies $\mathcal{M}$ with negligible advantage using $\mathcal{X}$, namely:

$$\mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \tag{5}$$

We next show the *Models Union Lemma (asymptotic-security)* lemma, which shows that if an adversary satisfies two models $\mathcal{M}$ and $\mathcal{M}'$, then $\mathcal{A}$ also satisfies the stronger model that is obtained by taking the union of $\mathcal{M}$ and $\mathcal{M}'$.

Lemma 2 (Models Union Lemma (asymptotic-security)).
For any set $\mathcal{X}$ of execution process operations and any two models $\mathcal{M}, \mathcal{M}'$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ with negligible advantage using $\mathcal{X}$, then $\mathcal{A}$ satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ with negligible advantage using $\mathcal{X}$, namely:

$$\left(\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \tag{6}$$

We next show the *Requirement-Model Monotonicity Lemma (asymptotic-security)* lemma, which shows that if a protocol satisfies a requirement under a weaker model, then it satisfies the same requirement under a stronger model (using the same operations set $\mathcal{X}$). This is true, because if we are assuming everything that is included in the stronger model, then we are assuming everything in the weaker model (by Lemma 1), which implies that the protocol satisfies the requirement for such adversaries.

Lemma 3 (Requirement-Model Monotonicity Lemma (asymptotic-security)).

For any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$, then $\mathcal{P}$ ensures $\mathcal{R}$ with negligible advantage under $\widehat{\mathcal{M}}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{poly} \mathcal{R} \tag{7}$$

5.2 Asymptotic-Security Requirement Modularity Lemmas

The requirement modularity lemmas prove 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 its (weaker) 'sub-requirements'.

The *Requirement Monotonicity Lemma (asymptotic-security)* lemma shows that if a protocol satisfies a stronger requirement $\widehat{\mathcal{R}}$, then it satisfies any requirement that is weaker than $\widehat{\mathcal{R}}$ (under the same model $\mathcal{M}$ and using the same operations set $\mathcal{X}$).

Lemma 4 (Requirement Monotonicity Lemma (asymptotic-security)).

For any set $\mathcal{X}$ of execution process operations, any model $\mathcal{M}$, and any requirements $\mathcal{R}$ and $\widehat{\mathcal{R}}$ such that $\mathcal{R} \subseteq \widehat{\mathcal{R}}$, if a protocol $\mathcal{P}$ satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ satisfies $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \tag{8}$$

Finally, the *Requirements Union Lemma (asymptotic-security)* lemma shows that if a protocol satisfies two requirements $\mathcal{R}$ and $\mathcal{R}'$, then it satisfies the stronger requirement that is obtained by taking the union of $\mathcal{R}$ and $\mathcal{R}'$ (under the same model $\mathcal{M}$ and operations set $\mathcal{X}$).

Lemma 5 (Requirements Union Lemma (asymptotic-security)).
For any set $\mathcal{X}$ of execution process operations, any models $\mathcal{M}$ and $\mathcal{M}'$, and any two requirements $\mathcal{R}$ and $\mathcal{R}'$, if a protocol $\mathcal{P}$ satisfies $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$ and satisfies $\mathcal{R}'$ with negligible advantage under $\mathcal{M}'$

using $\mathcal{X}$, then $\mathcal{P}$ satisfies the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv \mathcal{R} \cup \mathcal{R}'$ with negligible advantage under model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$, namely:

$$\left(\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M}',\,\mathcal{X}}_{poly} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{poly} \widehat{\mathcal{R}} \tag{9}$$

6 Using MoSS for Applied Specifications

In this section, we give a taste of how MoSS can be used to define applied security specifications, with realistic, non-trivial models and requirements. In Section 6.1, we discuss AuthBroadcast, a simple authenticated broadcasting protocol, which we use to demonstrate the use of MoSS's modularity lemmas. In Section 6.2 we discuss PKI schemes, which underlie the security of countless real-world applications, and show how MoSS enables rigorous requirements and models for PKI schemes. The definitions we show are only examples from [27], which present full specification and analysis of PKI schemes. The AuthBroadcast protocol is also not a contribution; we present it as an example.

6.1 AuthBroadcast: Authenticated Broadcast Protocol

In Appendix B, we present the AuthBroadcast protocol, a simple authenticated broadcast protocol that we developed and analyzed to help us fine-tune the MoSS definitions. AuthBroadcast 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. In this subsection, we present few details, as examples to the use of MoSS; in particular, AuthBroadcast addresses several aspects which do not exist in PKI scheme, such as shared-key initialization and confidentiality requirements. In particular, we define $\mathcal{M}_{\text{SecKeyShareInit}}$, a simple model for shared-key initialization. This shared-key model can be reused for specifications of many other tasks.

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, leveraging the modularity lemmas described in Section 5. Specifically:

1. We first show that AuthBroadcast ensures *authentication* of received messages, which we define as requirement $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$, under $\mathcal{M}_{\text{SecKeyShareInit}}$, the simple model for shared-key initialization.
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 the *freshness requirement* $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under the conjunction of $\pi^{\text{Drift}}_{\Delta_{clk}}$ (Algorithm 3) and $\pi_{\text{SecKeyShare}}$.

3. Finally, we show that $\mathsf{AuthBroadcast}$ also ensures *guaranteed bounded-delay delivery* of broadcast messages under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$, an even stronger model, that also assumes a bounded delay of communication. Specifically, we show that $\mathsf{AuthBroadcast}$ 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 $\mathsf{AuthBroadcast}$ also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_{\infty}}$ with negligible advantage under the stronger models in points 2 and 3, and similarly, that $\mathsf{AuthBroadcast}$ 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., $\mathsf{AuthBroadcast}$ also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ with negligible advantage under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$.

6.2 Specifications for PKI Scheme

PKI schemes are essential, and there were multiple incidents of vulnerabilities, and extensive research on improving PKI security. Provably-secure PKI schemes were presented in [12], however, these specifications did not cover aspects critical in practice, such as *timely revocation*, ensuring that relying parties will not be fooled into trusting a revoked certificate, not to mention more advanced properties such as transparency. In this subsection, we briefly discuss part of the realistic security specifications for PKI schemes, using MoSS, presented recently in [27].

Example models: $\mathcal{M}^{Drift}_{\Delta_{clk}}$ *and* $\mathcal{M}^{Com}_{\Delta_{com}}$. In [27], the authors define several models, covering the assumptions regarding the adversary capabilities, the environment (communication and synchronization) and the initialization, assumed by different PKI protocols. One of these is the bounded clock drift model $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$, which we defined in Section 3.3.

Let us give another example of a model assumed in [27]: the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model. As shown in Algorithm 7, $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ ensures reliable, bounded-delay delivery of messages sent.

Notice that both $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ and $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ are generic assumptions, which are assumed by many applied protocols. For example, $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ is also assumed by the $\mathsf{AuthBroadcast}$ protocol (subsection 6.1). Using MoSS, it often possible to reuse definitions of models, as well as of some 'generic' assumptions, among different works and tasks.

Example requirement: NReACC. Multiple PKI security requirements are defined in [27], from simple requirements like accountability, to more complex requirements like equivocation detection and prevention and transparency related requirements. We present one example: non-revocation accountability (NReACC), which intuitively requires that if a CA marks a certificate as 'non-revoked' at time t, while it actually earlier revoked this certificate, then this CA can be held

Algorithm 7 The $\pi^{\text{Com}}_{\Delta_{com}}$ $(T, params)$ Model

```
1: return (
2:     $\forall \hat{e}_S \in \{1, \dots, T.e - 1\}$:
3:         if ( $\exists$('send', $m, i_R) \in T.out[\hat{e}_S]$      ▷ If the output includes a send triple
4:             and $T.\tau[T.e] \geq T.\tau[\hat{e}_S] + \Delta_{com}$      ▷ And execution did not terminate yet after $\Delta_{com}$ real time
5:             and $T.ent[\hat{e}_S] \in T.\mathsf{N} - T.\mathsf{F}$ )      ▷ And the entity is honest
6:         then $\exists \hat{e}_R \in \{\hat{e}_S + 1, \dots, T.e\}$      ▷ Then there is a later event
7:             s.t. $T.\tau[\hat{e}_S] + \Delta_{com} \geq T.\tau[\hat{e}_R]$      ▷ Within $\Delta_{com}$ real time
8:             and $T.ent[\hat{e}_R] = i_R$      ▷ Where the entity is the intended recipient in the send triplet
9:             and $T.opr[\hat{e}_R]$ = 'Receive'      ▷ And which is a receive event
10:            and $T.inp[\hat{e}_R] = (m, T.ent[\hat{e}_S])$      ▷ And in which the entity receives the message from the sender
    )
```

accountable. In this way, NReACC, and other accountability requirements, deters CAs from misbehaving. NReACC is defined by the pseudocode in Algorithm 8.

Algorithm 8 Non-revocation accountability predicate NReACC(T)

```
1: $(\psi, \rho, pk, \iota) \leftarrow T.out_{\mathcal{A}}$      ▷ Certificate $\psi$, attribute attestation $\rho$, public key pk, and entity $\iota$
2: $AdvWins \leftarrow$ [
3:     $\iota \in T.\mathsf{N} - T.\mathsf{F}$ and      ▷ $\iota$ is an honest entity
4:     $\exists \hat{e}$ s.t. $T.ent[\hat{e}] = \iota$ and
         ('PubKey', $\psi.issuer, pk) \in T.out[\hat{e}]$ and      ▷ $\iota$ acknowledged that pk is the public key of $\psi.issuer$, i.e., the issuer of certificate $\psi$
5:     $\rho.attr$ = NREV and $\mathcal{P}$.WasValid$(\psi, pk, \rho)$ and      ▷ The adversary produced a valid attestation $\rho$ that $\psi$ was attested by $\psi.issuer$ as non-revoked on time $\rho.\tau$
6:     $\exists \bar{e}$ s.t. $T.ent[\bar{e}] = \psi.issuer$ and
                $T.opr[\bar{e}]$ = 'Revoke' and
                $T.inp[\bar{e}] = \psi$ and $T.\tau[\bar{e}] \leq \rho.\tau$      ▷ Yet, $\psi.issuer$ was asked to revoke $\psi$ before time $\rho.\tau$
7: ]
8: if $AdvWins$ then return $\perp$ else return $\top$
```

One of the benefits of defining separate requirements, is the ability to compare different schemes together according to the defined requirement. This is a non-trivial benefit, since comparing systems is not always easy when monolithic specifications are used. For example, see [27] for a comparison of PKI schemes based on the requirements they satisfy, where the requirements are specified using MoSS.

7 Two Extensions of MoSS

By now, we discussed and demonstrated a variety of specifications, which process an execution transcript T, outputted by executing the $\textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process discussed in Section 2.2. However, some specifications may need additional information beyond the transcript currently outputted by $\textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$, or require additional operations from the execution process. Trying to foresee that in advance, and have 'out-of-the-box' support to *any* imaginable scenario may fail, and/or result in unnecessarily complex framework. It seems better to have the built-in flexibility to support additional, non-trivial specifications as needed. In this section, we go over two important extensions and explain how MoSS can be (relatively) easy be extended to support them.

7.1 Concrete Security

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'). Let us explain how MoSS can support concrete security specifications.

First, we specify these bounds as *additional parameters* in *params* and provide these parameters to the adversary, i.e., denoted as $params.\mathcal{A}.bounds$. This allows model predicates to validate that the adversary does not *exceed* these bounds, with the exception of the adversary's runtime. Let us now explain how we can also enforce bounds on the adversary runtime.

To enforce bounds on the runtime of the adversary, we define a *compiler* algorithm $\mathsf{StepCount}$, whose input is algorithm $\mathcal{A}$, and whose output, $\mathsf{StepCount}(\mathcal{A})$, is an algorithm which outputs a pair: the same output as $\mathcal{A}$ would produce, and, in addition, the number of steps it took $\mathcal{A}$ to produce this output. This is a pretty standard 'trick' used in the theory of complexity.

Namely, for concrete security definitions, we use $\textbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$ instead of $\textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$. It remains to describe the operation of $\mathsf{StepCount}$.

Note that $\mathsf{StepCount}$ 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 $\mathsf{StepCount}$ 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 $\mathsf{StepCount}$, 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 $\mathsf{StepCount}$ 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 $\mathsf{StepCount}$ '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, $\mathsf{StepCount}$ first increments the $s_{\mathcal{A}}.\mathsf{StepCount}$ counter by the run-time of $\mathcal{A}$ in this specific invocation. Next, $\mathsf{StepCount}$ checks if $\mathcal{A}$ signaled termination of the execution process. When $\mathcal{A}$ signals termination (by returning $out_{\mathcal{A}} \neq \bot$), then $\mathsf{StepCount}$ 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[10]; of course, we still have $out_{\mathcal{A}} \neq \bot$ 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.

We now describe how concrete security may be enforced in MoSS using models and $\mathsf{StepCount}$ and adjust Definitions 1,2 and 3 to be compatible with concrete security definitions.

7.1.1 Enforcing Concrete Security via Models

We 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 denoted as $params.\mathcal{A}.bounds.maxCalls$, where each entry $params.\mathcal{A}.bounds.maxCalls[type][opr]$ 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 $\mathsf{StepCount}$, i.e., executing $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$. Then, we can use the π^{Bounds} model predicate (Algorithm 9) 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.

[10] 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 $\mathcal{A}$.

Algorithm 9 $\pi^{\text{Bounds}}(T,\, params)$ Predicate

```
1: return (
2:     $\forall\ type \in params.\mathcal{A}.bounds.maxCalls$:
3:         $\forall\ opr \in params.\mathcal{A}.bounds.maxCalls[type]$:      ▷ The number of calls to each operation with bounds is not exceeded
4:             $\left|\left\{ \hat{e} \;\middle|\; \begin{array}{l} \hat{e} \in \{1,\ldots,T.e\} \textbf{ and} \\ T.type[\hat{e}] = type \textbf{ and} \\ T.opr[\hat{e}] = opr \end{array} \right\}\right| \leq params.\mathcal{A}.bounds.maxCalls[type][opr]$
    and  $T.out_{\mathcal{A}}.\mathsf{StepCount} \leq params.\mathcal{A}.bounds.maxSteps$      ▷ The number of steps taken by $\mathcal{A}$ is not exceeded
    )
```

7.1.2 Definitions

Finally, in this section, we adjust Definitions 1, 2 and 3 for concrete security analysis. To that end, we first describe the difference in the use of the base function β in specifications in MoSS concrete security definitions, compared to the asymptotic definitions.

When using MoSS for concrete security analysis, for a specification (π, β), the function $\beta(params)$ is a *bound* on the probability of the adversary winning. Namely, there is no additional 'negligible' probability for the adversary to win, as we allowed in the asymptotic definitions. When $\mathcal{A}$ satisfies $\mathcal{M}$, for every specification in $\mathcal{M}$, the probability of $\mathcal{A}$ winning is *bounded* by the base function. Similarly, when $\mathcal{P}$ satisfies R under some model $\mathcal{M}$, for every $\mathcal{A}$ that satisfies $\mathcal{M}$ and every specification in R, the probability of $\mathcal{A}$ winning is *bounded* by the base function.

This implies that the base function is likely to differ when using MoSS for asymptotic analysis versus concrete security analysis; e.g., in asymptotic analysis, a specification $(\pi, 0)$ may be used, but in concrete security analysis, (π, β) may be used instead, where β is a (small) function that is not the zero constant function. This difference should be familiar to readers familiar with concrete-security definitions and results, e.g., [5]. Still, MoSS allows to use the same predicate π in both types of analysis.

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

Next, we define the advantage of an adversary $\mathcal{A}$ against a protocol $\mathcal{P}$ for a specification predicate π using execution operations $\mathcal{X}$ and $\mathsf{StepCount}$, as a function of *params*. This is the probability that $\pi(T, params) = \bot$, where T is a random resulting execution transcript $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)$.

Definition 4 (Advantage of $\mathcal{A}$ against $\mathcal{P}$ for specification predicate π using execution operations $\mathcal{X}$ and $\mathsf{StepCount}$). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X} \in PPT$ and let*

π be a specification predicate. The advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π using execution operations $\mathcal{X}$ and StepCount *is defined as:*

$$\epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \overset{def}{=} \Pr\begin{bmatrix}\pi\left(T, params\right) = \bot,\ where \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(params)\end{bmatrix} \tag{10}$$

We now give the concrete definition of model-satisfying adversary.

Definition 5 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$ and StepCount**).** *Let $\mathcal{A}, \mathcal{X} \in PPT$, and let $\mathcal{M}$ be a set of specifications, i.e., $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$. We say that* adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$ and StepCount*, denoted $\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}$, if for every protocol $\mathcal{P} \in PPT$, $params \in \{0,1\}^*$, and specification $(\pi, \beta) \in \mathcal{M}$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π using $\mathcal{X}$ and* StepCount *is bounded by $\beta(params)$, i.e.:*

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M} \overset{def}{=} \begin{bmatrix}\forall\ (\mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}) : \\ \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{bmatrix} \tag{11}$$

We also give the concrete definition of requirement-satisfying protocol.

Definition 6 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$ and StepCount**).** *Let $\mathcal{X} \in PPT$, and let $\mathcal{R}$ be a set of specifications, i.e., $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$. We say that* protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$ and StepCount*, denoted $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R}$, if for every PPT adversary $\mathcal{A}$ that satisfies $\mathcal{M}$ using execution operations $\mathcal{X}$ and* StepCount*, every parameters $params \in \{0,1\}^*$ where $params = (params.\mathcal{A}, params.\mathcal{P})$ and $|params.\mathcal{P}| \geq |params.\mathcal{A}|$, and every specification $(\pi, \beta) \in \mathcal{R}$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π using $\mathcal{X}$ and* StepCount *is bounded by $\beta(params)$, i.e:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \overset{def}{=} \begin{bmatrix}\begin{pmatrix}\forall\ \mathcal{A}\ s.t.\ \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}, \\ \forall params \in \{0,1\}^*,\ and\ \forall(\pi,\beta) \in \mathcal{R}\end{pmatrix} \\ \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{bmatrix} \tag{12}$$

7.2 Ensuring Reactive Polynomial 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 reactive (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 size of the original input.* 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 $\mathcal{A}$.

The output transcript of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ does not provide means to ensure polynomial runtime. However, $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ can be extended to ensure polynomial runtime, by adding two variables: $sLog[\cdot]$ and $maxLen[\cdot]$. $sLog[\cdot]$ stores the initial $|s_{\mathcal{A}}|$, the largest initial $|s_i|$ for over all $i \in \mathsf{N}$, and the initial $|s_{\mathcal{X}}|$. Additionally, $sLog[\cdot]$ stores the final sizes of $s_{\mathcal{A}}$, $s_{ent[\cdot]}$, and $s_{\mathcal{X}}$, for each iteration of the execution process. 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. We define the $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}} = (\pi^{\mathcal{A}}_{\text{polyGrow}}, 0)$ model, where the $\pi^{\mathcal{A}}_{\text{polyGrow}}$ predicate (Algorithm 10) restricts the size of each output of $\mathcal{A}$ in line 7 of $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ 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 13 to be at most $T.sLog[0][1]$ 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 $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$, so $\mathcal{A}$ itself defines this value at the beginning of the execution, but the value remains fixed for that particular execution. For the full details, see App. C.

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

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$:                                    ▷ For each event
3:         $\forall y \in \{T.ent[\hat{e}], T.opr[\hat{e}], T.type[\hat{e}], T.inp[\hat{e}], T.clk[\hat{e}], T.\tau[\hat{e}]\}$:   ▷ Each output of $\mathcal{A}$ from its first invocation
4:             $|y| \leq T.sLog[\hat{e}-1][1] + T.sLog[0][1]$                 ▷ Has size $\leq$ $T.sLog[0][1]$ plus the initial $|s_{\mathcal{A}}|$
5:         and $T.sLog[\hat{e}][1] \leq \max\{T.sLog[\hat{e}-1][1], |T.out[\hat{e}]|\} + T.sLog[0][1]$   ▷ And the final $|s_{\mathcal{A}}|$ is $\leq$ $T.sLog[0][1]$ plus the largest input size in the second invocation of $\mathcal{A}$
    )
```

8 Conclusions and Future Work

The MoSS framework enables modular specifications for applied cryptographic protocols, combining different models and requirements, each defined separately. As a result, MoSS allows comparison of protocols based on the requirements they satisfy and the models they assume. Definitions of specifications, and even of some (generic) requirements, may be reused across different protocols and problems. While obviously it takes some effort to learn MoSS, we found that the rewards of modularity and reusability justify the effort.

Future work includes the important challenges of (1) developing computer-aided mechanisms that support MoSS, e.g., 'translating' the modular MoSS specifications into a form supported by computer-aided proof tools, or developing computer-aided proof tools for MoSS, possibly using the modularity lemmas of section 5, and (2) extending the MoSS framework to support secure compositions.

Acknowledgements

We thank the anonymous reviewers for their insightful and constructive feedback; among other things, it helped us improve the definitions of models and requirements. We also thank Yuval Ishay, Sergio Rajsbaum, Juan Garay and Iftach Haitner for their comments and suggestions on earlier drafts of the paper. Part of the work was done while Prof. Syta had a visiting position in University of Connecticut. This work was partially supported by the Comcast Corporation. The opinions expressed are of the authors and not of their university or funding sources.

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. In: IEEE Symposium on Security and Privacy (2021)
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., Canetti, R., Krawczyk, H.: A modular approach to the design and analysis of authentication and key exchange protocols. IACR Cryptol. ePrint Arch **1998**, 9 (1998), `http://eprint.iacr.org/1998/009`
5. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: FOCS. pp. 394–403 (1997)
6. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) Advances in Cryptology—CRYPTO '93. Lecture Notes in Computer Science, vol. 773, pp. 232–249. Springer-Verlag (22–26 Aug 1993)
7. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: EUROCRYPT (2006)
8. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing (2008)
9. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible universal composability made simple. In: EUROCRYPT (2019)
10. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: IEEE Symposium on Foundations of Computer Science (2001)
11. Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: CRYPTO (2015)
12. Canetti, R., Shahaf, D., Vald, M.: Universally Composable Authentication and Key-exchange with Global PKI. Cryptology ePrint Archive, Report 2014/432 (2014), `https://eprint.iacr.org/2014/432`
13. CCITT, B.B.: Recommendations X. 509 and ISO 9594-8. Information Processing Systems-OSI-The Directory Authentication Framework (Geneva: CCITT) (1988)

14. Degabriele, J.P., Paterson, K., Watson, G.: Provable security in the real world. IEEE Security & Privacy **9**(3), 33–41 (2010)
15. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM **33**(4), 792–807 (Oct 1986)
16. Goldwasser, S., Micali, S.: Probabilistic Encryption. Journal of Computer and System Sciences **28**(2), 270–299 (1984)
17. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on computing (1988)
18. Hofheinz, D., Shoup, V.: Gnuc: A new universal composability framework. Journal of Cryptology **28**(3), 423–508 (2015)
19. Hofheinz, D., Unruh, D., Müller-Quade, J.: Polynomial runtime and composability. Journal of Cryptology **26**(3), 375–441 (2013)
20. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology pp. 1–124 (2020)
21. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Jun 2013). https://doi.org/10.17487/RFC6962
22. Leibowitz, H., Piotrowska, A.M., Danezis, G., Herzberg, A.: No Right to Remain Silent: Isolating Malicious Mixes. In: USENIX Security 19 (2019)
23. Lochbihler, A., Sefidgar, S.R., Basin, D., Maurer, U.: Formalizing constructive cryptography using crypthol. In: Computer Security Foundations (2019)
24. Maurer, U.: Constructive cryptography–a new paradigm for security definitions and proofs. In: Workshop on Theory of Security and Applications (2011)
25. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Computer Aided Verification (2013)
26. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
27. When PKI (finally) met provable security - full version. Anonymized, `https://sites.google.com/view/provablysecure/full`
28. Wikström, D.: Simplified universal composability framework. In: Theory of Cryptography Conference. Springer (2016)

A Illustration of the Execution Process

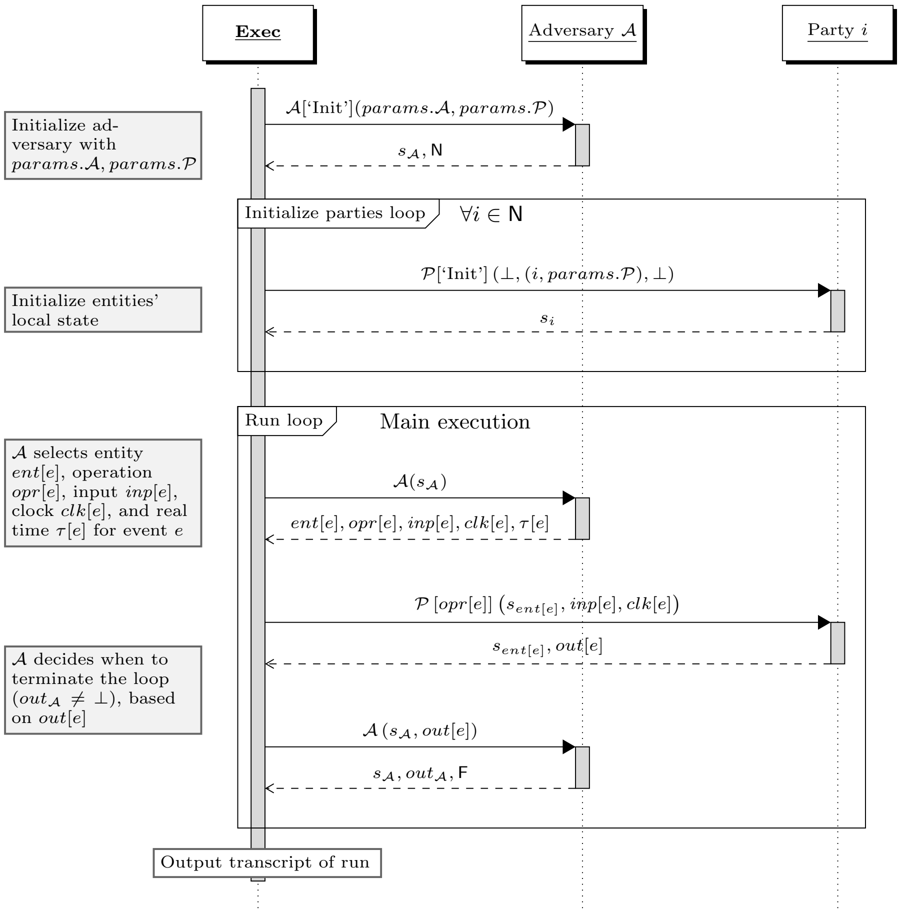
Exec
Adversary $\mathcal{A}$
Party i
$\mathcal{A}$['Init']$(params.\mathcal{A}, params.\mathcal{P})$
Initialize adversary with $params.\mathcal{A}, params.\mathcal{P}$
$s_{\mathcal{A}}, \mathsf{N}$
Initialize parties loop
$\forall i \in \mathsf{N}$
$\mathcal{P}$['Init'] $(\perp, (i, params.\mathcal{P}), \perp)$
Initialize entities' local state
s_i
Run loop
Main execution
$\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event e
$\mathcal{A}(s_{\mathcal{A}})$
$ent[e], opr[e], inp[e], clk[e], \tau[e]$
$\mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$
$s_{ent[e]}, out[e]$
$\mathcal{A}$ decides when to terminate the loop ($out_{\mathcal{A}} \neq \perp$), based on $out[e]$
$\mathcal{A}\,(s_{\mathcal{A}}, out[e])$
$s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}$
Output transcript of run

Fig. 3: A visual description of the MoSS's execution process.

B 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.

B.1 The AuthBroadcast Protocol

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 §B.5 for the formal definition of the authentication scheme along with its security definition.)

The AuthBroadcast protocol is a PPT algorithm with the operations:

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

where:

- 'Init' (Algorithm 16): initializes the local state of the entity.
- 'Key-Gen' (Algorithm 17): generates the shared authentication key and sends it to the rest of the entities.
- 'Sec-in' (Algorithm 18): receives a shared-key from another entity.
- 'Broadcast' (Algorithm 19): broadcasts an authenticated message.
- 'Receive' (Algorithm 20): receives incoming broadcast message, verify its authenticity and output it (to the application).

We first define the protocol's model specifications in Sec. B.2, the desired security requirements in Sec. B.3, and the formal definition of the protocol in Sec. B.4. We conclude this section with a formal security analysis of the AuthBroadcast protocol in Sec. B.6.

B.2 AuthBroadcast Model Specifications

We now define the $\mathcal{M}_{\mathsf{AuthBroadcast}}$, which we use to analyze the AuthBroadcast protocol under. The $\mathcal{M}_{\mathsf{AuthBroadcast}}$ model is defined as

$$\mathcal{M}_{\mathsf{AuthBroadcast}} = \mathcal{M}_{\text{SecKeyShareInit}} \ \wedge \ \mathcal{M}^{\text{Excl}}_{\mathcal{P}[\text{'Sec-in'}]} \ \wedge \ \mathcal{M}^{\text{Broadcast}}_{\Delta_{com}, \Delta_{clk}}$$

which we next define each of these sub-models.

The $\mathcal{M}_{\text{SecKeyShareInit}}$ model specification is defined as $\mathcal{M}_{\text{SecKeyShareInit}} = (\pi_{\text{SecKeyShare}}, 0)$, where the $\pi_{\text{SecKeyShare}}$ predicate (Algorithm 11) has two objectives. First, it ensures that only one entity securely shared a key during the protocol's execution. Second, before any 'Broadcast' or 'Receive' operation invoked on any entity $i \in \mathsf{N}$, the predicate ensures that i 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.

We emphasize that the reasoning this model is mainly 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.

Algorithm 11 $\pi_{\text{SecKeyShare}}(T, \mathit{params})$ Predicate

```
1: return ⊤ 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
   )
```

The $\mathcal{M}^{\text{Excl}}_{\mathcal{P}[\text{‘Sec-in’}]}$ model specification is defined as $\mathcal{M}^{\text{Excl}}_{\mathcal{P}[\text{‘Sec-in’}]} = (\pi^{\text{Excl}}_{\mathcal{P}[\text{‘Sec-in’}]}, 0)$, where the $\pi^{\text{Excl}}_{\mathcal{P}[\text{‘Sec-in’}]}$ predicate make sure that the adversary does not cause an entity to receive ‘fake’ securely shared values (using the ‘Sec-in’ operation). 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 12 $\pi^{\text{Excl}}_{\mathcal{P}[\text{‘Sec-in’}]}(T, \mathit{params})$ Predicate

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

The $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}}$ model specification $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}} = (\pi^{\text{Broadcast}}_{\Delta_{com}}, 0)$, where the $\pi^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ 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 13.

Algorithm 13 $\pi^{\text{Broadcast}}_{\Delta_{com}}(T, params)$ Predicate

```
1: return (
2:   $\forall \hat{e}_B \in \{1, \ldots, T.e - 1\}$:
3:       if $T.out[\hat{e}_B] = (\text{'Broadcast'}, m, timeSent, tag)$      ▷ If the output includes a broadcast message (with timestamp and tag)
4:           and $T.\tau[T.e] \geq T.\tau[\hat{e}_B] + \Delta_{com}$      ▷ And execution did not terminate yet after $\Delta_{com}$ real time
5:       then $\forall i \in \mathsf{N}$ s.t. $i \neq T.ent[\hat{e}_B]$:      ▷ Then for each entity except the broadcasting entity
6:           $\exists \hat{e}_R \in \{\hat{e}_B + 1, \ldots, T.e\}$      ▷ There is a later event
7:               and $T.\tau[\hat{e}_B] + \Delta_{com} \geq T.\tau[\hat{e}_R]$      ▷ Within $\Delta_{com}$ real time
8:               and $T.ent[\hat{e}_R] = i$      ▷ Where the entity is $i$
9:               and $T.opr[\hat{e}_R] = \text{'Receive'}$      ▷ And which is a receive event
10:              and $T.inp[\hat{e}_R] = (m, timeSent, tag)$      ▷ 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 14, 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 14 $\pi^{f \leq \Delta}_{\Delta_{com}, \Delta_{clk}}(T, params)$ Predicate

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

a conjunction of $\pi^{\text{Drift}}_{\Delta_{clk}}$, $\pi_{\text{SecKeyShare}}$, 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. B.6, we claim that the AuthBroadcast protocol satisfies $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ (described in Sec. B.3) 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.

B.3 AuthBroadcast 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 $\pi_{\mathsf{AuthComRcv}_\Delta}$ is a predicate is defined in Sec. 4.2 and where $\beta_{\mathsf{AuthComRcv}_\Delta}(params) = 2^{-params.\mathcal{P}.n}$. 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 15.

Algorithm 15 $\pi_{\mathsf{Broadcast}_{\Delta_{com}}}(T, params)$ Predicate

```
1: return (
2:   $\forall \hat{e}_B \in \{1, \ldots, T.e-1\}$ s.t.
3:       ($T.opr[\hat{e}_B] =$ 'Broadcast' and                ▷ For every 'Broadcast' event
4:       $T.\tau[T.e] \geq T.\tau[\hat{e}_B] + \Delta_{com}$):      ▷ That the execution did not terminate $\Delta_{com}$ real time after it
5:         $\forall i \in \mathsf{N}$ s.t. $i \neq T.ent[\hat{e}_B], \exists \hat{e}_R \in \{\hat{e}_B + 1, \ldots, T.e\}$ :   ▷ There is a later event for each entity except the broadcasting entity
6:             where $T.\tau[\hat{e}_B] + \Delta_{com} \geq T.\tau[\hat{e}_R]$      ▷ Within $\Delta_{com}$ real time
7:             and $T.out[\hat{e}_R] = (\text{'Receive'}, T.inp[\hat{e}_B].m)$   ▷ Where the broadcast message was received correctly
8:             and $T.ent[\hat{e}_R] = i$                          ▷ By the relevant entity
   )
```

B.4 AuthBroadcast Implementation

The AuthBroadcast protocol is a PPT algorithm with the following operations:

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

described in Algorithms 16-20. 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 16 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$[‘Init’]$(s, inp, clk)$

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

Algorithm 17 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$[‘Key-Gen’]$(s, inp, clk)$

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

Algorithm 18 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$[‘Sec-in’]$(s, inp, clk)$

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

Algorithm 19 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$[‘Broadcast’]$(s, inp, clk)$

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

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

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

B.5 Message Authentication Scheme

We now provide definitions for *message authentication scheme* and *unforgeable* message authentication scheme.

Definition 7. *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 8 (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.:*

$$\forall\ \mathcal{A} \in PPT, n \in \mathbb{N}, 1^\kappa : \quad max\{0, Pr[\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa) = \top] - 2^{-n}\} \in Negl(1^\kappa) \tag{14}$$

where $\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$ *is defined in Algorithm 21. 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 21 $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^\kappa)$

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

B.6 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 $\mathsf{AuthBroadcast}$ in several steps, where in each step, we prove that $\mathsf{AuthBroadcast}$ satisfies additional requirements - assuming increasingly stronger models:

1. We first show that $\mathsf{AuthBroadcast}$ ensures *authentication* of received messages under $\mathcal{M}_{\text{SecKeyShareInit}}$; note that $\mathcal{M}_{\text{SecKeyShareInit}}$ is a *generic* model - it may be reused for analysis of other shared-key protocols. Namely, we show that $\mathsf{AuthBroadcast}$ satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under model $\mathcal{M}_{\text{SecKeyShareInit}}$.
2. We then show that $\mathsf{AuthBroadcast}$ 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, $\mathsf{AuthBroadcast}$ 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 11).
3. Finally, we show that $\mathsf{AuthBroadcast}$ also ensures *guaranteed bounded-delay delivery* of broadcast messages under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$, an even stronger model, that also assumes a bounded delay of communication. Specifically, we show that $\mathsf{AuthBroadcast}$ 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 $\mathsf{AuthBroadcast}$ also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\infty}$ with negligible advantage under the stronger models in points 2 and 3, and similarly, that $\mathsf{AuthBroadcast}$ 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., $\mathsf{AuthBroadcast}$ also satisfies $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ with negligible advantage under $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com},\Delta_{clk}}$.

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 B.5. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

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

Where $\mathcal{M}_{\text{SecKeyShareInit}}$ is defined in Sec. B.2 and $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ is described in Sec. B.3.

claim 2. Let MAC be an asymptotically UF message authentication scheme, as defined in Appendix B.5. 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. B.3.

claim 3. Let MAC be a message authentication scheme, as defined in Appendix B.5. 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. B.2 and $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ is defined in Sec. B.3.

B.6.1 Proof of claim 1

In this section, we prove claim 1 of Theorem 1 - that $\mathsf{AuthBroadcast}$ 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 B.5. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

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

Where $\mathcal{M}_{\text{SecKeyShareInit}}$ is defined in Sec. B.2 and $\mathcal{R}_{\mathsf{AuthComRcv}_\Delta}$ is described in Sec. B.3.

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 $\mathcal{M}_{\text{SecKeyShareInit}}$ 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 $\mathcal{M}_{\text{SecKeyShareInit}}$ using execution operations $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M}_{\text{SecKeyShareInit}}$ 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 $\mathcal{M}_{\text{SecKeyShareInit}}$ 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 19) 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 20) 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 ('$\mathsf{Receive}$', 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 $x \overset{\mathbf{R}}{\leftarrow} \{0,1\}^n, tag \overset{\mathbf{R}}{\leftarrow} \{0,1\}^n$.

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

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^\kappa) \tag{15}$$

Where:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \Pr\begin{bmatrix}\pi_{\mathsf{AuthComRcv}_\infty}(T, params) = \bot \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)\end{bmatrix} - 2^{-params.\mathcal{P}.n} \tag{16}$$

We show now that Equation 15 implies Equation 17, as stated in Lemma 6 below. Note that the difference between Eq. 15 and Eq. 17 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 6. *Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}_{SecKeyShareInit}$ satisfying Equation 15 for some $params \in \{0,1\}^*$. Then holds:*

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^\kappa) \tag{17}$$

Where:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_\infty},\mathsf{StepCount}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}},\mathcal{X}}(params) \equiv \Pr\begin{bmatrix}\pi_{\mathsf{AuthComRcv}_\infty}(T, params) = \bot \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)\end{bmatrix} - 2^{-params.\mathcal{P}.n} \tag{18}$$

Proof of Lemma 6.

From Equation 15, $\mathcal{A}$ 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. 19, 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. 20, 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, with non-negligible advantage, $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \,||\, timeSent) = tag$.

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 $\mathsf{OVer}(m \,||\, timeSent, tag) = \top$ (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 17.

$\square$

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 B.5. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined as in Sec. 2.3). Then:

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

Where $\mathcal{M}_{\text{SecKeyShareInit}}$ is defined in Sec. B.2 and $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ is described in Sec. B.3.

Proof. Suppose that $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ under $\mathcal{M}$. Then Equation 15 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. 21), 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 6, Equation 17 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 ('$\mathsf{Receive}$', m), yet there is no previous 'Broadcast' event $\hat{e}_{\text{B}}$ where the input is m. Whenever this is the case, $\mathcal{A}'$ outputs $T.inp[\hat{e}_{\text{R}}].m \mid\mid T.inp[\hat{e}_{\text{R}}].timeSent, T.inp[\hat{e}_{\text{R}}].tag$. 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.:

$$\max\{0, \Pr\left[\mathbf{Exp}^{\text{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^{\kappa}) = \top\right] - 2^{-params.\mathcal{P}.n}\} \notin Negl(1^{\kappa}) \tag{19}$$

Therefore, MAC is not asymptotically UF (according to Def. 8 in App. B.5). $\square$

B.6.2 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. B.6.1), 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 B.5. 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. B.3.

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 19) 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 20) to:

   $$\textbf{and } \mathsf{OVer}(m \mid\mid timeSent, tag) = \top$$

3. $\mathcal{A}'$ executes $T \leftarrow \textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathsf{AuthBroadcast}}}(params)$, 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 $(\text{'}\mathsf{Receive}\text{'}, m)$, 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 $x \overset{\mathbf{R}}{\leftarrow} \{0,1\}^n, tag \overset{\mathbf{R}}{\leftarrow} \{0,1\}^n$.

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:

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{20}$$

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

$$\epsilon^{\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \Pr\begin{bmatrix} \pi_{\mathsf{AuthComRcv}_{\Delta}}(T, params) = \bot \ : \\ T \leftarrow \textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params) \end{bmatrix} - 2^{-params.\mathcal{P}.n} \tag{21}$$

We show now that Equation 20 implies Equation 22, as stated in Lemma 7 below. Note that the difference between Eq. 20 and Eq. 22 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 7. *Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}$ satisfying Equation 20 for some $params \in \{0,1\}^*$. Then holds:*

$$\epsilon^{\mathcal{R}_{\mathit{AuthComRcv}_{\Delta}}}_{\mathcal{A},\overline{\mathit{AuthBroadcast}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{22}$$

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

$$\epsilon^{\mathcal{R}_{\mathit{AuthComRcv}_{\Delta}},\mathsf{StepCount}}_{\mathcal{A},\overline{\mathit{AuthBroadcast}},\mathcal{X}}(params) \equiv \Pr\begin{bmatrix} \pi_{\mathit{AuthComRcv}_{\Delta}}(T, params) = \bot \ : \\ T \leftarrow \textbf{Exec}^{\mathcal{X}}_{\mathcal{A},\overline{\mathit{AuthBroadcast}}}(params) \end{bmatrix} - 2^{-params.\mathcal{P}.n} \tag{23}$$

Proof of Lemma 7.

From Equation 20, $\mathcal{A}$ 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. 19, 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. 20, 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 $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \mid\mid timeSent) = tag$. 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 $\mathsf{MAC}_{params.\mathcal{P}.n}(k, m \mid\mid timeSent) = tag$ 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 $\mathsf{OVer}(m \mid\mid timeSent, tag) = \top$ 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 22.

□

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 B.5. 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 $\pi = \pi_{\mathrm{SecKeyShare}} \wedge \pi^{\mathrm{Drift}}_{\Delta_{clk}}$. Then:

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

Where $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ is described in Sec. B.3.

Proof. Suppose that $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not satisfy $\mathcal{R}_{\mathsf{AuthComRcv}_{\Delta}}$ under $\mathcal{M}$. Then Equation 20 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. 21), 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 7, Equation 22 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 ('$\mathsf{Receive}$', 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 $\hat{e}_{\mathrm{R}}$. 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.:

$$\max\{0, \Pr\left[\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(n, 1^{\kappa}) = \top\right] - 2^{-params.\mathcal{P}.n}\} \notin Negl(1^{\kappa}) \qquad (24)$$

Therefore, MAC is not asymptotically UF (according to Def. 8 in App. B.5). $\square$

B.6.3 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 $\mathcal{M}^{\mathrm{Broadcast}}_{\Delta_{com},\Delta_{clk}}$. The claim is restated below.

claim 3. Let MAC be a message authentication scheme, as defined in Appendix B.5. Let $\mathcal{X}$ be {'Sec-in'} (where the 'Sec-in' $\mathcal{X}$-operation is defined in Sec. 2.3). Then:

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

Where $\mathcal{M}^{\mathrm{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ is defined in Sec. B.2 and $\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}$ is defined in Sec. B.3.

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 $\mathcal{M}^{\mathrm{Broadcast}}_{\Delta_{com},\Delta_{clk}}$ 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:

$$\epsilon^{\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \notin Negl(params.\mathcal{P}.1^{\kappa}) \tag{25}$$

Where

$$\epsilon^{\mathcal{R}_{\mathsf{Broadcast}_{\Delta_{com}}}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}},\mathcal{X}}(params) \equiv \Pr\left[\begin{array}{l}\pi_{\mathsf{Broadcast}_{\Delta_{com}}}(T, params) = \bot \ : \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathsf{AuthBroadcast}^{\mathsf{MAC}}}(params)\end{array}\right] \tag{26}$$

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. 13) and $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}.\Delta$ (see Algorithm 14).

Recall that, according to Alg. 19, 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. 20, 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 7 in App. B.5, implying that MAC is not a message authentication scheme. □

C Ensuring Polynomial-Time Runtime

Our definitions of models and requirements, were given for PPT adversary $\mathcal{A}$. Namely, $\mathcal{A}$'s run-time is bounded to a polynomial in its input length. However, what is the length of the input to $\mathcal{A}$? During the execution, $\mathcal{A}$ is invoked many times, with different inputs. In the first call, when $\mathcal{A}$ is initialized, its input is *params*; but what is the size of subsequent inputs?

The inputs to the adversary $\mathcal{A}$ in round i of the execution, consist of the output of the protocol in round i, and of the state of the adversary from the previous round. Let us focus on the size of the adversary's state. Intuitively, we may expect that state cannot grow, but the execution process does not impose such restriction. Therefore, we have to consider that the state of $\mathcal{A}$ may grow (polynomially) in each round. However, this implies that over i rounds, the state may grow *exponentially*; e.g., if the state 'only' doubles in each round, then after i rounds it grows by 2^i. Even if the number of rounds is linear in the size of *params*, the state - and time available to the adversary - may grow exponentially!

Furthermore, it is the *adversary* who determines the number of rounds. What prevents the adversary from running for an exponential number of rounds?

This problem existed in other cryptographic execution processes, e.g., the one used by UC [10], and has been addressed in different ways. We now show how we can address it in MoSS. The solution isn't trivial, but we believe it is simpler than solutions we have seen.

C.1 Polynomial-growth PPT Algorithms

Recall that the problem stems from the fact that a single execution may involve multiple invocations of an algorithm (specifically, $\mathcal{A}$ and $\mathcal{P}$). Intuitively, we want to prevent an algorithm from increasing the length of its outputs 'too much', so that when used as inputs, they will cause super-polynomial increase in the input size, and therefore also time-complexity and output size.

To ensure this, we bound the *growth* of the output size of an algorithm after a sequence of invocations, by a polynomial in size of the *initial input params*. We refer to this class of algorithms as Polynomial-growth PPT (PgPPT[11]) algorithms, and define it below.

However, before we define Polynomial-growth PPT (PgPPT) algorithms, we need to define what we mean by a *sequence of invocations*. The definition, which follows, ensures that the algorithm maintains its *state* across the sequence of invocations; this allows the algorithm to maintain the bound on the growth of its output, as a function of the initial input *params*. This is the 'normal' way for invoking the same algorithm multiple times, maintaining its state by the calling process, as done by the MoSS execution process.

In this definition, we refer to the input parameters to an invocation Inv of an algorithm, as an array $Inv.in[i]$, where $Inv.in[0]$ is the input state. Similarly, we refer to the outputs of the invocation Inv as an array $Inv.out[i]$, where $Inv.out[0]$ is the output state.

Definition 9 (Sequence of invocations of an algorithm $\mathcal{Z}$). *Let $\mathcal{Z}$ be an algorithm, and let $Inv = \{Inv_1, Inv_2, \ldots\}$ be a sequence of pairs of arrays,*

[11] The name is in honor of PgP and Phil Zimmerman.

$Inv_i = (Inv_i.in[\cdot], Inv_i.out[\cdot])$, for the inputs and output, respectively. We say that Inv is a sequence of invocations of $\mathcal{Z}$, if:

1. *Every tuple is an event of $\mathcal{Z}$, namely: $(\forall i) Inv_i.out = \mathcal{Z}(Inv_i.in)$[12].*
2. *The states are consecutive: $(\forall i > 1)$ $Inv_i.in[0] = Inv_{i-1}.out[0]$.*
3. *The first state has the special initial state 'Init': $Inv_1.in[0] =$'Init'.*
4. *'Init' is never an output state: $(\forall i)$ $Inv_i.out[0] \neq$'Init'.*

Executions of the MoSS execution model, include a (separate) sequence of invocations for the adversary $\mathcal{A}$, for each entity $i \in \mathsf{N}$, and (for $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$) also for the $\mathcal{X}$ algorithm (defining extensions to the execution process).

In Polynomial-growth PPT (PgPPT) algorithms, the size of all outputs is bounded by the size of the largest input, plus the at most the output of the initialization step. The definition uses $|Inv_i.in[\cdot]|_{Max}$ and $|Inv_i.out[\cdot]|_{Max}$ to denote the maximal (longest) value in the array $in[\cdot]$ (respectively, $out[\cdot]$).

Definition 10 (Polynomial-growth PPT (PgPPT) algorithm). *We say that a PPT algorithm $\mathcal{Z}$ is a* polynomial-growth PPT (PgPPT) algorithm *if there exists a polynomial $p(\cdot)$ such that for every sequence of invocations Inv of $\mathcal{Z}$ and for every $i > 1$ s.t. $Inv_i \in Inv$, holds:*

$$|Inv_i.out[\cdot]|_{Max} \leq |Inv_i.in[\cdot]|_{Max} + p\left(|Inv_1.out[\cdot]|_{Max}\right) \tag{27}$$

C.2 The PgPPT lemma

We now present and prove the *PgPPT Lemma*, which, intuitively, shows polynomial time complexity for polynomial-length executions of Polynomial-growth PPT Algorithms, where the polynomials are in the length of the input parameters (*params*).

For the lemma to hold, we need two assumptions, which we define by corresponding model (and predicates): the $\mathcal{M}_{\text{polySteps}} = (\pi_{\text{polySteps}}, 0)$ model, which limits the number of events in an execution, and the $\mathcal{M}_{\text{1-Init}} = (\pi_{\text{1-Init}}, 0)$ model, which verifies that $\mathcal{A}$ does not re-initialize $\mathcal{P}$ or $\mathcal{X}$ during an execution.

The $\pi_{\text{polySteps}}$ predicate verifies 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 state $s_{\mathcal{A}}$ when it is initially output by the adversary in line 1 of Algorithm 2.

Algorithm 22 The $\pi_{\text{polySteps}}$ $(T,\ params)$ Predicate

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

The $\pi_{\text{1-Init}}$ predicate verifies that both $\mathcal{P}$ and $\mathcal{X}$ are only initialized in lines 2 and 3 of the execution process, respectively.

[12] If $\mathcal{Z}$ is randomized, the condition is that $Inv_i.out$ is a possible outcome of $\mathcal{Z}(Inv_i.in)$.

Algorithm 23 The $\pi_{\text{1-Init}}$ $(T, \textit{params})$ Predicate

```
1: return ( $\forall \hat{e} \in \{1, \ldots, T.e\}$ : $opr[\hat{e}] \neq$ 'Init' )    ▷ $\mathcal{A}$ does not choose 'Init' operation in any event
```

Before stating and proving the lemma, we first prove a polynomial bound on the length of the states in each iteration of the execution, provided that the predicates hold.

Claim. Let $\mathcal{X}$, $\mathcal{A}$, and $\mathcal{P}$ be PgPPT algorithms. Then there exist polynomials $\{p_k\}$ such that for any execution $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$ where $\pi_{\text{polySteps}}(T) = \top$ and $\pi_{\text{1-Init}}(T) = \top$, at the end of every iteration, $k \in \{0, \ldots, T.e\}$ of this execution, the lengths of $|s_{\mathcal{A}}|$, $|s_{\mathcal{X}}|$, and $|s_{ent[k]}|$ are all bounded by $p_k(|params|)$.

Proof. Since $\pi_{\text{polySteps}}(T) = \top$, we know that the number of iterations, which we denote k_{Max}, is bounded by a polynomial applied to the length of the output of $\mathcal{A}$ after its initial invocation (in which $\mathcal{A}$ is initialized). Since $\mathcal{A}$ is in PPT, then k_{Max} is bounded by a polynomial in the length of $\mathcal{A}$, i.e., by a polynomial in $|params|$.

The proof is by induction on the iteration number, k, from 0 to k_{Max}. Specifically, we will prove that after k iterations, the length of the outputs is bounded by a polynomial $p_k(|secparams|)$. The claim follows.

Initially (i.e., for $k = 0$), the claim holds immediately from the fact that each of these algorithms is PPT. Assume, therefore, that the statement is true up to iteration k and show that it will be true for iteration $k + 1$ as well.

In the $(k+1)^{\text{th}}$ iteration, when $\mathcal{A}$ is invoked in line 7, its input consists only of its state at the end of the previous iteration (k), i.e., its length is bounded by $p_k(|params|)$. Since $\mathcal{A}$ is PgPPT, and the execution process initializes $\mathcal{A}$ only once (in iteration $k = 0$), it follows that all of its outputs are bounded by the length of its input, plus a polynomial in $|params|$.

The inputs to an entity $i \in \mathsf{N}$ running $\mathcal{P}$, consist of the existing state of that entity - which is the output of $\mathcal{P}$ in some earlier iteration - and of inputs defined by the adversary. So, from the induction hypothesis and discussion in previous paragraph, both are bounded by polynomials in $|params|$. Since $\mathcal{P}$ is also PgPPT, and since $\pi_{\text{1-Init}}(T) = \top$, i.e., entities in N are initialized only in the initial iteration, it follows that the outputs of i are also bounded by a polynomial in $|params|$.

Alternatively, if line 9 is executed, then, similarly, the largest input to $\mathcal{X}$ is bounded by a polynomial in $|params|$; and hence, again similarly, the output must also be bounded by a polynomial in $|params|$.

Finally, the same argument shows that in line 13, the input to $\mathcal{A}$ is bounded by a polynomial in $|params|$, and hence, for same reasoning, the outputs of $\mathcal{A}$ are also bounded by a polynomial in $|params|$. Namely, the statement holds. □

We can now prove the PgPPT lemma.

Lemma 8 (PgPPT lemma). *Let $\mathcal{X}$, $\mathcal{A}$, and $\mathcal{P}$ be PgPPT algorithms. If $\mathcal{A}$ satisfies $\mathcal{M}_{polySteps}$ and $\mathcal{M}_{1\text{-}Init}$, then for any execution $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$, the total running time of $\mathcal{A}$ and the total running time of $\mathcal{P}$ are both bounded by a polynomial in params, except for negligible probability.*

Proof. Since $\mathcal{A}$ satisfies $\mathcal{M}_{\text{polySteps}} = (\pi_{\text{polySteps}}, 0)$ and $\mathcal{M}_{\text{1-Init}} = (\pi_{\text{1-Init}}, 0)$, then there is only a negligible probability that $\pi_{\text{polySteps}}(T) = \bot$ or $\pi_{\text{1-Init}}(T) = \bot$ (since the sum of negligible functions is negligible). Thus, we only need to show that if both $\pi_{\text{polySteps}}(T) = \top$ and $\pi_{\text{1-Init}}(T) = \top$, then the total running time of $\mathcal{A}$ and the total running time of $\mathcal{P}$ are both bounded by a polynomial in *params*. Basically, this follows from the claim above; details omitted. □

C.3 Polynomial Bound Without Polynomial-growth PPT Assumption

Lemma 8 applies when we know that $\mathcal{A}$, $\mathcal{P}$, and $\mathcal{X}$ are all PgPPT algorithms. We can clearly make sure that $\mathcal{X}$ is PgPPT, since we define it; but it is more tricky to verify that an arbitrary given protocol $\mathcal{P}$ is PgPPT- and surely preferable *not* to assume that the adversary is PgPPT. In this subsection, we show that we can indeed avoid these assumptions. This requires:

- $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}$, an additional extension to the execution process, only requiring two more lines.
- an additional *model*, $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$, which essentially replaces the *assumption* that $\mathcal{A}$ is PgPPT.
- an additional *requirement*, $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$, which essentially replaces the *assumption* that $\mathcal{P}$ is PgPPT.

We now present each of these components.

C.3.1 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 24) builds on the $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$ execution process discussed in Section 2.2, but it has an additional variable called $maxLen[\cdot]$, with an entry for each event of the execution process, and it adds more entries to the $sLog[\cdot]$ variable. As in $\mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}$, $sLog[0]$ holds the initial $|s_{\mathcal{A}}|$, the largest initial $|s_i|$ for over all $i \in \mathsf{N}$, and the initial $|s_{\mathcal{X}}|$. Additionally, the $sLog[\cdot]$ variable includes $sLog[\cdot][1]$, $sLog[\cdot][2]$, and $sLog[\cdot][3]$ for each iteration of the execution process, which store the final sizes of $s_{\mathcal{A}}$, $s_{ent[\cdot]}$, and $s_{\mathcal{X}}$, respectively, 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. Line 9 is added to save the $maxLen[e]$ value for each event e, line 16 is added to store the sizes of the output states in $sLog[e]$, and $maxLen[\cdot]$ is added to the execution transcript T in line 18.

C.3.2 The $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ Requirement

We define the $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$ requirement as $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}} = (\pi^{\mathcal{P}}_{\text{polyGrow}}, 0)$, where $\pi^{\mathcal{P}}_{\text{polyGrow}}$ is the predicate shown in Algorithm 25. The predicate restricts the size of each

Algorithm 24 Extendible Adversary-Driven Execution Process $\mathbf{Exec}^{\mathcal{X}PPT}_{\mathcal{A},\mathcal{P}}(params)$

```
1:  $(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}[\text{'Init'}](params.\mathcal{A}, params.\mathcal{P})$    ▷ Initialize adversary with params.A, params.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:  $s_{\mathcal{X}} \leftarrow \mathcal{X}[\text{'Init'}](params)$    ▷ Initial exec state
4:  $sLog[0] \leftarrow (|s_{\mathcal{A}}|, \max_{i\in\mathsf{N}}\{|s_i|\}, |s_{\mathcal{X}}|)$
5:  $e \leftarrow 0$    ▷ Initialize loop's counter
6:  repeat
7:      $e \leftarrow e + 1$    ▷ Advance the loop counter
8:      $(ent[e], opr[e], type[e], inp[e], clk[e], \tau[e]) \leftarrow \mathcal{A}(s_{\mathcal{A}})$    ▷ A selects entity ent[e], operation opr[e], input inp[e], clock clk[e], and real time τ[e] for event e
9:      $\boldsymbol{maxLen[e] \leftarrow \max\{|opr[e]|, |inp[e]|, |clk[e]|, |s_{ent[e]}|\}}$
10:     if $type[e] = \text{'}\mathcal{X}\text{'}$ then    ▷ If A chose to invoke an operation from X.
11:         $(s_{\mathcal{X}}, s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{X}[opr[e]](s_{\mathcal{X}}, s_{ent[e]}, inp[e], clk[e], ent[e])$
12:     else    ▷ A chose to invoke an operation from P.
13:         $(s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}[opr[e]](s_{ent[e]}, inp[e], clk[e])$
14:     end if
15:     $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}(s_{\mathcal{A}}, out[e])$    ▷ A decides when to terminate the loop ($out_{\mathcal{A}} \neq \perp$), based on out[e]
16:     $\boldsymbol{sLog[e] \leftarrow (|s_{\mathcal{A}}|, |s_{ent[e]}|, |s_{\mathcal{X}}|)}$    ▷ Save lengths of $s_{\mathcal{A}}$, $s_{ent[e]}$, and $s_{\mathcal{X}}$ to return as part of T
17: until $out_{\mathcal{A}} \neq \perp$
18: $T \leftarrow (out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], type[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], sec\text{-}out[\cdot][\cdot], sLog[\cdot], maxLen[\cdot])$
19: Return $T$    ▷ Output transcript of run
```

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 24, so $\mathcal{P}$ defines this value at the beginning of the execution, but the value remains fixed for that particular execution.

Algorithm 25 The $\pi^{\mathcal{P}}_{\text{polyGrow}}$ $(T,\, params)$ Predicate

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$:                          ▷ For each event
3:         if $T.type[\hat{e}] = `\mathcal{X}'$:                  ▷ If $\mathcal{A}$ chose an $\mathcal{X}$ operation, then the size of each output of $\mathcal{X}$ is $\leq T.sLog[0][3]$ plus the size of the largest input to $\mathcal{X}$
4:             $\forall y \in \{T.sLog[\hat{e}][3], T.sLog[\hat{e}][1], T.out[\hat{e}], T.sec\text{-}out[\hat{e}][\cdot], T.ent[\hat{e}]\}$:
5:                 $|y| \leq \max(T.maxLen[\hat{e}], T.sLog[\hat{e}-1][3], T.ent[\hat{e}]) + T.sLog[0][3]$
6:         else:                                          ▷ If $\mathcal{A}$ did not choose an $\mathcal{X}$ operation, then the size of each output of $\mathcal{P}$ is $\leq T.sLog[0][2]$ plus the size of the largest input to $\mathcal{P}$
7:             $\forall y \in \{T.sLog[\hat{e}].s_{ent}, T.out[\hat{e}], T.sec\text{-}out[\hat{e}][\cdot]\}$:
8:                 $|y| \leq T.maxLen[\hat{e}] + T.sLog[0][2]$
   )
```

C.3.3 Using $\mathcal{M}^{\mathcal{A}}_{\textbf{polyGrow}}$ and $\mathcal{R}^{\mathcal{P}}_{\textbf{polyGrow}}$ Instead of Assuming Polynomial-growth PPT

First, the following claim proves a polynomial bound over the length of the states in each iteration.

Claim. Let $\mathcal{A}$, and $\mathcal{P}$ be PPT algorithms and $\mathcal{X}$ be a PgPPT algorithm. Then there exist polynomials $\{p_k\}$ such that for any execution $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$ where $\pi^{\mathcal{A}}_{\text{polyGrow}}(T) = \top$, $\pi_{\text{polySteps}}(T) = \top$, $\pi_{\text{1-Init}}(T) = \top$, and $\pi^{\mathcal{P}}_{\text{polyGrow}}(T) = \top$, at the end of every iteration k of this execution, the lengths of $|s_{\mathcal{A}}|$, $|s_{\mathcal{X}}|$, and $|s_{ent[k]}|$ are all bounded by $p_k(|params|)$.

The proof is omitted as it is very similar to the proof of the previous claim.

We now show that if $\mathcal{A}$ satisfies a model $\mathcal{M}$ which combines $\mathcal{M}^{\mathcal{A}}_{\text{polyGrow}}$, $\mathcal{M}_{\text{polySteps}}$, and $\mathcal{M}_{\text{1-Init}}$, i.e., $\mathcal{M} = (\pi^{\mathcal{A}}_{\text{polyGrow}} \wedge \pi_{\text{polySteps}} \wedge \pi_{\text{1-Init}}, 0)$, and $\mathcal{P}$ satisfies $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}}$, then the total running times of $\mathcal{A}$ and $\mathcal{P}$ during executions of the MoSS execution process will be polynomial (except for negligible probability).

Lemma 9 (Model-based Polytime). *Let $\mathcal{A}, \mathcal{P} \in PPT$ and $\mathcal{X} \in PgPPT$. Assume $\mathcal{A}$ satisfies $\mathcal{M} = (\pi^{\mathcal{A}}_{polyGrow} \wedge \pi_{polySteps} \wedge \pi_{1\text{-}Init}, 0)$, i.e., $\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}$ and $\mathcal{P}$ satisfies $\mathcal{R}^{\mathcal{P}}_{polyGrow}$ under $\mathcal{M}$, i.e., $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{poly} \mathcal{R}^{\mathcal{P}}_{polyGrow}$. Then, during the execution of $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}(params)$, the total running time of $\mathcal{A}$ and the total running time of $\mathcal{P}$ are both bounded by a polynomial in params, except for negligible probability (in $|params|$).*

Proof. The proof is similar to the proof of Lemma 8, with some differences, which we mention here.

The first difference is that we consider $\mathcal{A}$ and $\mathcal{P}$ which are PPT but are not necessarily PgPPT; however, $\mathcal{A}$ satisfies $\mathcal{M} = (\pi^{\mathcal{A}}_{\text{polyGrow}} \wedge \pi_{\text{polySteps}} \wedge \pi_{\text{1-Init}}, 0)$ and $\mathcal{P}$ satisfies $\mathcal{R}^{\mathcal{P}}_{\text{polyGrow}} = (\pi^{\mathcal{P}}_{\text{polyGrow}}, 0)$ under $\mathcal{M}$. This implies that there is only negligible probabilty that for an execution transcript T of $\mathbf{Exec}^{\mathcal{X},PPT}_{\mathcal{A},\mathcal{P}}(params)$, any one of $\pi^{\mathcal{A}}_{\text{polyGrow}}(T)$, $\pi_{\text{polySteps}}(T)$, $\pi_{\text{1-Init}}(T)$, and $\pi^{\mathcal{P}}_{\text{polyGrow}}(T)$ is false (since the sum of negligible functions is negligible). Thus, consider any execution transcript T for which all four of $\pi^{\mathcal{A}}_{\text{polyGrow}}(T)$, $\pi_{\text{polySteps}}(T)$, $\pi_{\text{1-Init}}(T)$, and $\pi^{\mathcal{P}}_{\text{polyGrow}}(T)$ are true, which happens with overwhelming probability. We just need to show that in this case, the total running time of $\mathcal{A}$ and the total running time of $\mathcal{P}$ are both bounded by a polynomial in $params$. Since we already found a polynomial bound, the rest of the proof is the same as for Lemma 8.

□

D Additional Specification Predicates

D.1 Examples of Models and Model Predicates

D.1.1 Threshold Security model $\mathcal{M}^{|\mathsf{F}|\leq f}$: up to f Faults

We now define the popular *threshold security* fault model, which allows up to some threshold number of the entities to be faulty, i.e., controlled by the adversary. 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 $|\mathsf{N}|$. The base function for this model is 0, i.e., we do not allow any probability for more than f faults. Namely, $\mathcal{M}^{|\mathsf{F}|\leq f} = (\pi^{|\mathsf{F}|\leq f}, 0)$, where we define the threshold security predicate $\pi^{|\mathsf{F}|\leq f}$ in Algorithm 26.

Algorithm 26 $\pi^{|\mathsf{F}|\leq f}$ $(T,\ params)$ Predicate

```
1: return (
2:     ($|T.\mathsf{F}| \leq f\,(|T.\mathsf{N}|)$)                              ▷ Max size of $T.\mathsf{F}$ is not exceeded
3:     and $\forall \hat{e} \in \{1, \ldots, T.e\}$ :                          ▷ For each event
4:         if $T.opr[\hat{e}] \in$ {'Get-state', 'Set-state', 'Set-output'} and   ▷ If the operation means the ad-
             $T.type[\hat{e}] =$ '$\mathcal{X}$'                                     versary controls the entity
5:         then $T.ent[\hat{e}] \in T.\mathsf{F}$                                ▷ Then entity is in $T.\mathsf{F}$
    )
```

The $\pi^{|\mathsf{F}|\leq f}$ predicate allows the adversary three 'fault' operations, which allow a variety of attack models; additional models can further limit the attack model. Let us briefly explain the three fault operations and give some examples for their use for different attack models:

'Get-state': return the state of the entity, including any secret/private keys. If we add a model only 'Get-state' faults, we get the classical *honest-but-curious* adversary model.

'Set-output': force specific output from the entity. This can allow, in particular, the classical *byzantine fault model*, where the attacker controls all outputs from the entity.

'Set-state': set a specific state for the entity. If we add a model allowing only 'Set-state' faults and only upon the very beginning of the execution, we obtain the *self-stabilization* model.

To enforce the model predicate, the predicate simply ensures that the 'Get-state', 'Set-state', and 'Set-output' operations are applied only to entities in $T.\mathsf{F}$, and that $|T.\mathsf{F}| \leq f(|\mathsf{N}|)$.

D.1.2 $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$: authentic-sender, bounded-delay communication

We next present $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$, an authentic-sender, bounded-delay communication model. As with many models, the base-function is zero, i.e., $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}} = (\pi^{\text{AuthCom}}_{\Delta_{com}}, 0)$. 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:

$$\pi^{\text{AuthCom}}_{\Delta_{com}}(T, params) = \pi^{\text{Com}}_{\Delta_{com}}(T, params) \wedge \pi^{\text{AuthCom-rcv}}_{\Delta_{com}}(T, params) \quad (28)$$

Note that these two predicates can also be used to define corresponding models: $\mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}} = (\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}, 0)$ and $\mathcal{M}^{\text{Com}}_{\Delta_{com}} = (\pi^{\text{Com}}_{\Delta_{com}}, 0)$, for cases where only one of the two assumptions is required.

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

The authentic-sender property ($\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate) implies that $inp[\hat{e}_{\text{R}}].i_{\text{S}}$ indeed sent this message to $ent[\hat{e}_{\text{R}}]$, during some previous event $\hat{e}_{\text{S}} < \hat{e}_{\text{R}}$. The $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate is shown in Algorithm 27.

Algorithm 27 $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ (T, params) Predicate

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

D.1.3 The Δ-precise Wakeup Model $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$

We next present the $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$ model; again, the base-function is zero, i.e., $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}} = (\pi^{\text{Wake-up}}_{\Delta_{clk}}, 0)$. We refer to $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$ as the Δ-precise Wakeup Model.

$\pi^{\text{Wake-up}}_{\Delta_{clk}}$ 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 (*'Sleep'*, 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 28.

Algorithm 28 $\pi^{\text{Wake-up}}_{\Delta_{clk}}$ $(T,\, params)$ Predicate

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$:                         ▷ For each event $\hat{e}$
3:         if ( $(\text{'}Sleep\text{'}, x) \in T.out[\hat{e}]$             ▷ If the output includes a ('Sleep', x) tuple
4:             and $T.\tau[T.e] \geq T.\tau[\hat{e}]+x+\Delta_{clk}$ )   ▷ And execution did not terminate yet after $x + \Delta_{clk}$ real time
5:         then $\exists \hat{e}' \in \{\hat{e}+1, \ldots, T.e\}$          ▷ Then there is a later event
6:             s.t. $|T.\tau[\hat{e}'] - T.\tau[\hat{e}] - x| \leq \Delta_{clk}$   ▷ With real time x greater than at $\hat{e}$ (within $\Delta_{clk}$)
7:             and $T.ent[\hat{e}'] = T.ent[\hat{e}]$                ▷ In which the entity is the same as in $\hat{e}$
8:             and $T.opr[\hat{e}'] =$ 'Wake-up'                    ▷ And the operation is 'Wake-up'
    )
```

D.1.4 Secure shared-keys Initialization ($\mathcal{M}_{\textbf{SecKeyShareInit}}$) Model

Sometimes, especially when analyzing shared-key protocols, it may be useful to *assume* that the entities securely share keys before communicating with each other - i.e., to assume that the keys are correctly shared without the interference of the adversary. For this purpose, we define the model $\mathcal{M}_{\text{SecKeyShareInit}}$, shown in Algorithm 29, which ensures that values (keys) are securely shared (using the *sec-out* mechanism in the execution process) before entities send or receive messages. Specifically, before any 'Send' or 'Receive' event, the sender of the message must have securely shared values with the receiver, and the receiver of the message must have securely shared values with the sender. We assume that the operation 'Sec-channel-setup' is used to share these values.

A similar model for secure (authenticated) initialization of public keys is defined in [27].

D.2 Requirement Predicates

D.2.1 The Verified Attribution Generic Requirement $\mathcal{R}_{\mathcal{VAS}}$

The output of many protocols may include *attributable statements*. An *attributable statement* is a tuple (m, σ, i), where m is a string to which we refer as a *statement*, $i \in \mathsf{N}$ 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.,

Algorithm 29 $\pi_{\text{SecKeyShare}}(T,\ params)$ Predicate

```
1: return $\top$ if (
2:     $\forall \hat{e}'' \in \{1, \ldots, T.e\}$ s.t. $T.opr[\hat{e}''] =$ 'Send' or $T.opr[\hat{e}''] =$ 'Receive':
                                                      $\triangleright$ For each message 'Send' event and each message 'Receive' event
3:         if $T.opr[\hat{e}''] =$ 'Send'                 $\triangleright$ Determine the sender and receiver of the message
           then $i_{\mathrm{S}}, i_{\mathrm{R}} \leftarrow T.ent[\hat{e}''], T.inp[\hat{e}''].i_{\mathrm{R}}$
           else if $T.opr[\hat{e}''] =$ 'Receive'
           then $i_{\mathrm{S}}, i_{\mathrm{R}} \leftarrow T.inp[\hat{e}''].i_{\mathrm{S}}, T.ent[\hat{e}'']$
4:         $\exists\ \hat{e}, \hat{e}'$
               s.t. $1 \leq \hat{e} < \hat{e}' < \hat{e}''$      $\triangleright$ Previously (before the 'Send' or 'Receive' event)
5:             and $T.opr[\hat{e}] =$ 'Sec-channel-setup'   $\triangleright$ Keys were securely sent
               and $T.ent[\hat{e}] = i_{\mathrm{S}}$              $\triangleright$ From the sender of the message
               and $T.sec\text{-}out[\hat{e}][i_{\mathrm{R}}] \neq \bot$    $\triangleright$ To the receiver of the message
6:             and $T.opr[\hat{e}'] =$ 'Sec-in'             $\triangleright$ And they were securely received
               and $T.ent[\hat{e}'] = i_{\mathrm{R}}$             $\triangleright$ By the receiver of the message
               and $T.inp[\hat{e}'] = \hat{e}$
7:         and $\exists\ \hat{e}, \hat{e}'$
               s.t. $1 \leq \hat{e} < \hat{e}' < \hat{e}''$      $\triangleright$ And also previously (before the 'Send' or 'Receive' event)
8:             and $T.opr[\hat{e}] =$ 'Sec-channel-setup'   $\triangleright$ Keys were securely sent
               and $T.ent[\hat{e}] = i_{\mathrm{R}}$              $\triangleright$ From the receiver of the message
               and $T.sec\text{-}out[\hat{e}][i_{\mathrm{S}}] \neq \bot$    $\triangleright$ To the sender of the message
9:             and $T.opr[\hat{e}'] =$ 'Sec-in'             $\triangleright$ And securely received
               and $T.ent[\hat{e}'] = i_{\mathrm{S}}$             $\triangleright$ By the sender of the message
               and $T.inp[\hat{e}'] = \hat{e}$
     )
```

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 $(\text{'}\mathsf{public\ key}\text{'}, pk) \in out[\hat{e}]$, in some event $\hat{e}$; namely, we use '$\mathsf{public\ key}$' 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 § D.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):

$$V_{ka}(i, pk, T) = \{\exists \hat{e} \ s.t. \ T.ent[\hat{e}] = i \ \wedge (\text{'}\mathsf{public\ key}\text{'}, pk) \in T.out[\hat{e}]\} \qquad (29)$$

We now define the Verified Attribution of Statements Requirement $\mathcal{R}_{\mathcal{VAS}}$; the base-function is zero, i.e., $\mathcal{R}_{\mathcal{VAS}} = (\pi_{\mathsf{VAS}}, \prime)$. 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 $(\text{'}\mathsf{signed}\text{'}, m)$ as part of its output, i.e., $(\text{'}\mathsf{signed}\text{'}, m) \in out[\hat{e}]$. Since this is *always* done, whenever the protocol signs a message, we will *not* explicitly include the $(\text{'}\mathsf{signed}\text{'}, m)$ 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. 29). 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 30.

Algorithm 30 Verifiable Attribution of Statements Predicate $\pi_{\mathsf{VAS}}(T, params)$

```
1: $(m, \sigma, i, pk) \leftarrow T.out_{\mathcal{A}}$
2: return $\neg($
3:    $i \in T.\mathsf{N} - T.\mathsf{F}$                                   ▷ i is an honest entity
4:    and $\mathcal{S}.\mathsf{Ver}(pk, m, \sigma) = \top$                    ▷ m was signed by the owner of pk
5:    and $V_{ka}(i, pk, T) = \top$                         ▷ i identified pk as its public key
6:    and $\nexists \hat{e}$ s.t.: $T.ent[\hat{e}] = i$
          and ('signed', $m) \in T.out[\hat{e}]$           ▷ Yet, i never indicated that it signed m
    $)$
```

D.2.2 The No False Positives Generic Requirement

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 [22]. 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. We also define a third requirement predicate, *no false positives*, which is simply the conjunction of the other two; we omit the simple definition of this combined requirement. In the rest of this subsection, we present the *non-*

frameability requirement (and predicate); and in the next subsection, we present the *no false accusation* requirement predicate.

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 31, returns $\bot$ if the adversary was able to output a Proof of Misbehavior for an honest entity, and $\top$ otherwise.

Algorithm 31 Non-frameability Predicate $\pi_{\mathsf{NF}}(T, params)$

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

D.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 App. D 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, $\pi_{\mathsf{NFA}}(T, params)$ 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 32.

Algorithm 32 No False Accusations Predicate $\pi_{\mathsf{NFA}}(T, params)$

```
1: return ¬(
2:     $T.ent[T.e] \in T.\mathsf{N} - T.\mathsf{F}$                              ▷ $T.ent[T.e]$ is an honest entity
3:     and $\exists j \in T.\mathsf{N} - T.\mathsf{F}, x$ s.t. $(\mathrm{IA}, j, x) \in T.out[T.e]$   ▷ $T.ent[T.e]$ 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 33.

Algorithm 33 Use First-Accuse Time Predicate $\pi_{\mathsf{UFAT}}(T, params)$

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

E Proofs of the Asymptotic-Security Modularity Lemmas

In this section, we show proofs of the asymptotic-security modularity lemmas presented in Sec. 5.

E.1 Proofs of the Asymptotic-Security Modularity Lemmas

Lemma 1 (Model Monotonicity Lemma (asymptotic-security)).

For any set $\mathcal{X}$ of execution process operations, for any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if an adversary $\mathcal{A}$ satisfies $\widehat{\mathcal{M}}$ with negligible advantage using $\mathcal{X}$ then $\mathcal{A}$ satisfies $\mathcal{M}$ with negligible advantage using $\mathcal{X}$, namely:

$$\mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \tag{5}$$

Proof. Using Def. 2, the left side means that:

$$\begin{aligned}&(\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi,\beta) \in \widehat{\mathcal{M}}) :\\ &\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

Since $\mathcal{M} \subseteq \widehat{\mathcal{M}}$:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi,\beta) \in \mathcal{M}) :\\ &\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

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

Lemma 2 (Models Union Lemma (asymptotic-security)).

For any set $\mathcal{X}$ of execution process operations and any two models $\mathcal{M}, \mathcal{M}'$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ with negligible advantage using $\mathcal{X}$, then $\mathcal{A}$ satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ with negligible advantage using $\mathcal{X}$, namely:

$$\left(\mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{poly} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{poly} \widehat{\mathcal{M}} \tag{6}$$

Proof. By Def. 2, the left side means that:

$$\begin{aligned}&(\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi,\beta) \in \mathcal{M}) :\\ &\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\\ &\wedge (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi,\beta) \in \mathcal{M}') :\\ &\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

Since $\widehat{\mathcal{M}} = \mathcal{M} \cup \mathcal{M}'$, we have:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi,\beta) \in \widehat{\mathcal{M}}) :\\ &\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

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

Lemma 3 (Requirement-Model Monotonicity Lemma (asymptotic-security)).
For any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$, then $\mathcal{P}$ ensures $\mathcal{R}$ with negligible advantage under $\widehat{\mathcal{M}}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{poly} \mathcal{R} \tag{7}$$

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

$$\begin{aligned}(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

Since $\mathcal{M}$ is weaker than $\widehat{\mathcal{M}}$, then by Lemma 1:

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

Therefore, we have:

$$\begin{aligned}\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

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

Lemma 4 (Requirement Monotonicity Lemma (asymptotic-security)).
For any set $\mathcal{X}$ of execution process operations, any model $\mathcal{M}$, and any requirements $\mathcal{R}$ and $\widehat{\mathcal{R}}$ such that $\mathcal{R} \subseteq \widehat{\mathcal{R}}$, if a protocol $\mathcal{P}$ satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ satisfies $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \tag{8}$$

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

$$\begin{aligned}(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \widehat{\mathcal{R}}) :\\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

Since $\mathcal{R} \subseteq \widehat{\mathcal{R}}$:

$$\begin{aligned}\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params) + Negl(|params|)\end{aligned}$$

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

Lemma 5 (Requirements Union Lemma (asymptotic-security)).

For any set $\mathcal{X}$ of execution process operations, any models $\mathcal{M}$ and $\mathcal{M}'$, and any two requirements $\mathcal{R}$ and $\mathcal{R}'$, if a protocol $\mathcal{P}$ satisfies $\mathcal{R}$ with negligible advantage under $\mathcal{M}$ using $\mathcal{X}$ and satisfies $\mathcal{R}'$ with negligible advantage under $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{P}$ satisfies the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv \mathcal{R} \cup \mathcal{R}'$ with negligible advantage under model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$, namely:

$$\left(\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{poly} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M}',\,\mathcal{X}}_{poly} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{poly} \widehat{\mathcal{R}} \tag{9}$$

Proof. Since $\widehat{\mathcal{M}}$ is stronger than $\mathcal{M}$ and stronger than $\mathcal{M}'$, then by Lemma 3:

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

Using Def. 3, this means that:

$$\begin{aligned}
&(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\
&\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params) + Negl(|params|)\\
&\wedge (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}') :\\
&\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params) + Negl(|params|)
\end{aligned}$$

Since $\widehat{\mathcal{R}} = \mathcal{R} \cup \mathcal{R}'$, we have:

$$\begin{aligned}
&\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\text{poly}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \widehat{\mathcal{R}}) :\\
&\qquad \epsilon^{\pi}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params) + Negl(|params|)
\end{aligned}$$

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

F Concrete-Security Modularity Lemmas

In Sec. 5, we presented the asymptotic-security modularity lemmas. We now show the corresponding concrete-security modularity lemmas (in App. F.1 and F.2) and their proofs (in App. F.3), which use the concrete-security definitions from Sec. 7.1.

F.1 Concrete-Security Model Modularity Lemmas

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 lemma is *Concrete-security weaker model satisfaction.* It shows that if an adversary $\mathcal{A}$ satisfies a stronger model $\widehat{\mathcal{M}}$, then $\mathcal{A}$ also satisfies any model that is weaker than $\widehat{\mathcal{M}}$.

Lemma 10 (Concrete-security weaker model satisfaction).

For any set $\mathcal{X}$ of execution process operations, for any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if an adversary $\mathcal{A}$ satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$ and StepCount*, then $\mathcal{A}$ satisfies $\mathcal{M}$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M} \tag{30}$$

We next show the *Concrete-security stronger model satisfaction* lemma, which shows that if an adversary satisfies two models $\mathcal{M}$ and $\mathcal{M}'$, then $\mathcal{A}$ also satisfies the stronger model that is obtained by taking the union of $\mathcal{M}$ and $\mathcal{M}'$.

Lemma 11 (Concrete-security stronger model satisfaction).

For any set $\mathcal{X}$ of execution process operations and any two models $\mathcal{M}, \mathcal{M}'$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ using $\mathcal{X}$ and StepCount*, then $\mathcal{A}$ satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\left(\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}} \tag{31}$$

We next show the *Concrete-security requirement satisfaction under stronger model* lemma, which shows that if a protocol satisfies a requirement under a weaker model, then it satisfies the same requirement under a stronger model (using the same operations set $\mathcal{X}$). This is true, because if we are assuming everything that is included in the stronger model, then we are assuming everything in the weaker model (by Lemma 10), which implies that the protocol satisfies the requirement for such adversaries.

Lemma 12 (Concrete-security requirement satisfaction under stronger model).

For any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$ and StepCount*, then $\mathcal{P}$ ensures $\mathcal{R}$ under $\widehat{\mathcal{M}}$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\mathcal{P} \models^{\mathcal{M}, \mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}}, \mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \tag{32}$$

F.2 Concrete-Security Requirement Modularity Lemmas

The requirement modularity lemmas prove 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 its (weaker) 'sub-requirements'.

The *Concrete-security weaker requirement satisfaction* lemma which shows that if a protocol satisfies a stronger requirement $\widehat{\mathcal{R}}$, then it satisfies any requirement that is weaker than $\widehat{\mathcal{R}}$ (under the same model $\mathcal{M}$ and using the same operations set $\mathcal{X}$).

Lemma 13 (Concrete-security weaker requirement satisfaction).

For any set $\mathcal{X}$ of execution process operations, any model $\mathcal{M}$, and any requirements $\mathcal{R}$ and $\widehat{\mathcal{R}}$ such that $\mathcal{R} \subseteq \widehat{\mathcal{R}}$, if a protocol $\mathcal{P}$ ensures the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$ and $\mathsf{StepCount}$*, then $\mathcal{P}$ ensures $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and* $\mathsf{StepCount}$*, namely:*

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \tag{33}$$

Finally, the *Concrete-security stronger requirement satisfaction* lemma shows that if a protocol satisfies two requirements $\mathcal{R}$ and $\mathcal{R}'$, then it satisfies the stronger requirement that is obtained by taking the union of $\mathcal{R}$ and $\mathcal{R}'$ (under the same model $\mathcal{M}$ and operations set $\mathcal{X}$).

Lemma 14 (Concrete-security stronger requirement satisfaction).

For any set $\mathcal{X}$ of execution process operations, any models $\mathcal{M}$ and $\mathcal{M}'$, and any two requirements $\mathcal{R}$ and $\mathcal{R}'$, if a protocol $\mathcal{P}$ ensures $\mathcal{R}$ under $\mathcal{M}$ and ensures $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$ and $\mathsf{StepCount}$*, then $\mathcal{P}$ ensures the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv \mathcal{R} \cup \mathcal{R}'$ under model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$ and* $\mathsf{StepCount}$*, namely:*

$$\left(\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M}',\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{R}} \tag{34}$$

F.3 Proofs of the Concrete-Security Modularity Lemmas

Lemma 10 (Concrete-security weaker model satisfaction).

For any set $\mathcal{X}$ of execution process operations, for any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if an adversary $\mathcal{A}$ satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$ and $\mathsf{StepCount}$*, then $\mathcal{A}$ satisfies $\mathcal{M}$ using $\mathcal{X}$ and* $\mathsf{StepCount}$*, namely:*

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M} \tag{30}$$

Proof. Using Def. 5, the left side means that:

$$\begin{gathered}(\forall\, \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \widehat{\mathcal{M}}) : \\ \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{gathered}$$

Since $\mathcal{M} \subseteq \widehat{\mathcal{M}}$:

$$\begin{gathered}\Rightarrow (\forall\, \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}) : \\ \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{gathered}$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}$, according to Def. 5. □

Lemma 11 (Concrete-security stronger model satisfaction).

For any set $\mathcal{X}$ of execution process operations and any two models $\mathcal{M}, \mathcal{M}'$, if an adversary $\mathcal{A}$ satisfies both $\mathcal{M}$ and $\mathcal{M}'$ using $\mathcal{X}$ and StepCount*, then $\mathcal{A}$ satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\left(\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}} \tag{31}$$

Proof. By Def. 5, the left side means that:

$$\begin{aligned}&(\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}) :\\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\\ &\wedge (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}') :\\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{aligned}$$

Since $\widehat{\mathcal{M}} = \mathcal{M} \cup \mathcal{M}'$, we have:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \widehat{\mathcal{M}}) :\\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{aligned}$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}}$, according to Def. 5. □

Lemma 12 (Concrete-security requirement satisfaction under stronger model).

For any models $\mathcal{M}$ and $\widehat{\mathcal{M}}$ such that $\mathcal{M} \subseteq \widehat{\mathcal{M}}$, if a protocol $\mathcal{P}$ ensures a requirement $\mathcal{R}$ under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$ and StepCount*, then $\mathcal{P}$ ensures $\mathcal{R}$ under $\widehat{\mathcal{M}}$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \tag{32}$$

Proof. By Def. 6, the left side means that:

$$\begin{aligned}&(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{aligned}$$

Since $\mathcal{M}$ is weaker than $\widehat{\mathcal{M}}$, then by Lemma 10:

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M}$$

Therefore, we have:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) :\\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \leq \beta(params)\end{aligned}$$

Which is the definition of $\mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R}$, according to Def. 6. □

Lemma 13 (Concrete-security weaker requirement satisfaction).

For any set $\mathcal{X}$ of execution process operations, any model $\mathcal{M}$, and any requirements $\mathcal{R}$ and $\widehat{\mathcal{R}}$ such that $\mathcal{R} \subseteq \widehat{\mathcal{R}}$, if a protocol $\mathcal{P}$ ensures the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$ and StepCount*, then $\mathcal{P}$ ensures $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \tag{33}$$

Proof. Using Def. 6, the left side means that:

$$\begin{aligned}&(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \widehat{\mathcal{R}}) : \\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params)\end{aligned}$$

Since $\mathcal{R} \subseteq \widehat{\mathcal{R}}$:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \mathcal{R}) : \\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params)\end{aligned}$$

Which is the definition of $\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R}$, according to Def. 6. □

Lemma 14 (Concrete-security stronger requirement satisfaction).

For any set $\mathcal{X}$ of execution process operations, any models $\mathcal{M}$ and $\mathcal{M}'$, and any two requirements $\mathcal{R}$ and $\mathcal{R}'$, if a protocol $\mathcal{P}$ ensures $\mathcal{R}$ under $\mathcal{M}$ and ensures $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$ and StepCount*, then $\mathcal{P}$ ensures the 'combined' (stronger) requirement $\widehat{\mathcal{R}} \equiv \mathcal{R} \cup \mathcal{R}'$ under model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ using $\mathcal{X}$ and* StepCount*, namely:*

$$\left(\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M}',\,\mathcal{X}}_{\mathsf{StepCount}} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{R}} \tag{34}$$

Proof. Since $\widehat{\mathcal{M}}$ is stronger than $\mathcal{M}$ and stronger than $\mathcal{M}'$, then by Lemma 12:

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

Using Def. 6, this means that:

$$\begin{aligned}&(\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \mathcal{R}) : \\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params) \\ &\wedge (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \mathcal{R}') : \\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params)\end{aligned}$$

Since $\widehat{\mathcal{R}} = \mathcal{R} \cup \mathcal{R}'$, we have:

$$\begin{aligned}&\Rightarrow (\forall\ \mathcal{A} \text{ s.t. } \mathcal{A} \models^{\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{M}},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \widehat{\mathcal{R}}) : \\ &\qquad \epsilon^{\pi,\mathsf{StepCount}}_{\mathcal{A},\mathcal{P},\mathcal{X}}(params) \le \beta(params)\end{aligned}$$

Which is the definition of $\mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{\mathsf{StepCount}} \widehat{\mathcal{R}}$, according to Def. 6. □