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 [11,28] (e.g., 'ideal functionality' in UC) or games [8,30], are *monolithic*, combining together different aspects of protocol requirements, environment and assumptions. Such security specifications are complex, error-prone, and foil reusability, modular analysis and incremental design.

We present the *Modular Security Specifications (MoSS) framework*, which cleanly separates the *security requirements* (goals) which a protocol should achieve, from the *models* (assumptions) under which each 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 or satisfying different sets of requirements. MoSS is flexible and extendable, e.g., it can support both *asymptotic* and *concrete* definitions for security. So far, we confirmed 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.

This article is based on an earlier article: Herzberg A., Leibowitz H., Syta E., Wrótniak S. (2021) MoSS: Modular Security Specifications Framework. In: Malkin T., Peikert C. (eds) Advances in Cryptology – CRYPTO 2021. CRYPTO 2021. Lecture Notes in Computer Science, vol 12827. Springer, Cham., ©IACR 2021 https://doi.org/10.1007/978-3-030-84252-9_2.

Provable security has its roots in the seminal works rigorously proving security for constructions of cryptographic primitives, such as signature schemes [18], encryption schemes [17] and pseudorandom functions [16]. 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 also extended to applied cryptographic protocols, with seminal works such as [4, 7]. After repeated discoveries of serious vulnerabilities in 'intuitively designed' protocols [15], 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. In response, we present the MoSS framework.

MoSS: Modular Security Specifications. In MoSS, a *security specification* includes a set of *models* (assumptions) and specific *requirements* (goals); models and requirements are defined using *predicates* and probability functions. By defining each model and requirement separately, we allow modularity, standardization and reuse. This modularity is particularly beneficial for applied protocols, due to their high number of requirements and models; see Figure 1.

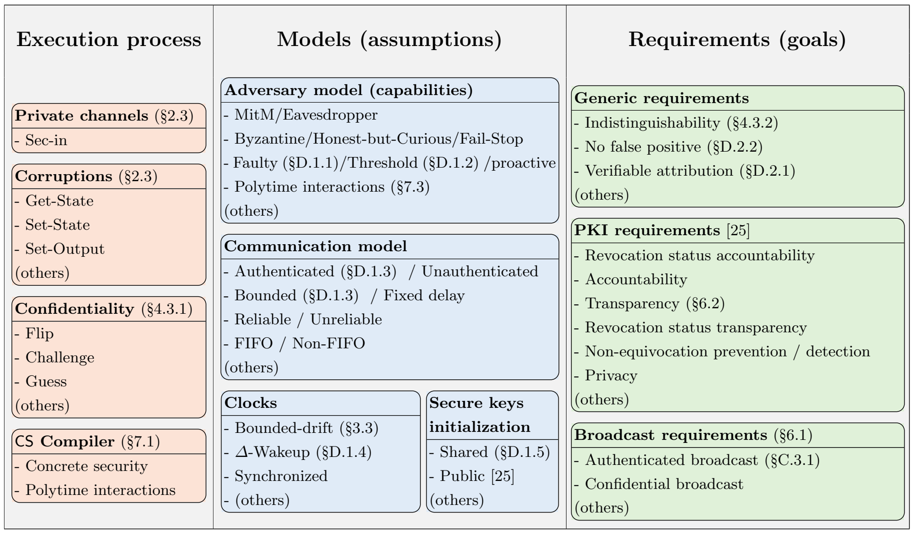

Fig. 1: The MoSS framework allows security to be specified *modularly*, i.e., 'à la carte', with respect to a set of individually-defined models (assumptions), requirements (properties/goals) and even operations of the execution process. Models, requirements and operations defined in this paper or in [19, 25] are marked accordingly. Many models, and some ('generic') requirements, are applicable to different types of protocols.

MoSS also includes a well-defined *execution process* (Figure 2 and Algorithm 1), as necessary for provable security. For simplicity, the 'core' execution process is simple, and supports modular extensions, allowing support for some specific features which are not always needed. Let us now discuss each of these three components of MoSS in more detail.

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, or bounded drift for the clock synchronization model. Often, a model can be reused in many works, since, in MoSS, each model is defined independently of other models and of requirements, as one or more pairs of a small predicate ('program') and a probability function. This approach facilitates the reuse of models and also makes it easier to write, read and compare different works. For example, many protocols, for different tasks, use the same clock and communication models, e.g., synchronous communication and clocks. At the same time, protocols for the same task may use different models, e.g., bounded delay communication and bounded drift clocks.

Requirements refer to properties or goals which a protocol aims for. 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 has a well-defined execution process (see Figure 2 and Algorithm 1) which takes as input a protocol to execute, an adversary, parameters and a set of *execution operations*. The execution operations allow customized extensions of the execution process, i.e., they enhance the basic execution process with operations which may not always be required. We use these additional operations to define specifications such as indistinguishability, shared-key initialization and entity corruptions.

Related work. A significant amount of work in applied cryptography is informally specified, with specifications presented as a textual list of assumptions (models) and goals (requirements). Obviously, this informal approach does not facilitate provable security. For provable security, there are two main approaches for defining security specifications: simulation-based and game-based.

The *simulation-based approach*, most notably Universal Composability (UC) [11, 12], typically defines security as indistinguishability between executions of the given protocol with the adversary, and executions of an 'ideal functionality', which blends together the model and requirements, with a *simulator*. There are multiple extensions and alternatives to UC, such as iUC, GNUC, IITM and simplified-UC [10, 21, 23, 31], and other simulation-based frameworks such as constructive cryptography (CC) [27, 28] and reactive systems [1]. Each of these variants defines a specific, fixed execution model. An important reason for the

Approach	**Specifications**			**Multiple specifications**	**Prov.-secure composition**
	Exec Process	**Models**	**Requirements**		
Informal	-	List	List	Yes	No
Game-based	Game per goal; models are part of game			Yes	No
Simulation-based	Fixed	Indistinguishable from Ideal Functionality		No	Yes
MoSS	Extensible	List	List	Yes	No

Table 1: A comparison of different approaches to security specifications. An execution process defines executions (runs). A protocol aims to satisfy certain *requirements* assuming certain *models*. Simulation-based specifications, such as UC [12], ensure *provably-secure composition* of protocols but do not allow one protocol to meet multiple separately-defined specifications. Some tasks, e.g. zero-knowledge, may only have simulation-based specifications.

popularity of the simulation-based approach is its support for *secure composition* of protocols; another reason is the fact that some important tasks, e.g., zero-knowledge (ZK), seem to require simulation-based definitions. However, for many tasks, especially applied tasks, game-based definitions are more natural and easier to work with.

The *game-based approach* [8, 20, 30] is also widely adopted, especially among practitioners, due to its simpler, more intuitive definitions and proofs of security. In this approach, each requirement is defined as a *game* between the adversary and the protocol. The game incorporates the models, the execution process, and the specific requirement (e.g., indistinguishability). However, the game-based approach does have limitations, most notably, there is no composition theorem for game-based specifications and it may be inapplicable to tasks such as zero-knowledge proofs and multi-party computation.

Both 'game-based' and 'simulation-based' security specifications are *monolithic*: an ideal functionality or a game, combining security requirements with different aspects of the model and the execution process. Even though different requirements and models are individually presented in their informal descriptions, the designers and readers have to validate directly that the formal, monolithic specifications correctly reflect the informal descriptions.

Such monolithic specifications are not a good fit for analysis of applied protocols, which have 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*.

We began this work after trying to write simulation-based as well as game-based specifications for PKI schemes, which turned out to be impractical given the complexity of realistic modeling aspects; this motivated us to develop modular security specifications, i.e., MoSS.

In Table 1, we compare MoSS to game-based and simulation-based security specifications. The advantage of MoSS is its *modularity*; a security specification consists of one or more *models*, one or more *requirements* and, optionally, some execution process operations. Each model and requirement is defined independently, as one or more pairs of a small *predicate* (which is, typically, a simple program) and a probability function. Models are often applicable to different tasks, and some requirements are generic and apply to multiple tasks. This modular approach allows to reuse models and requirements, which makes it easier to write, understand and compare specifications. For example, in Appendix C, we present a simplified instance of an authenticated-broadcast protocol assuming (well-defined) bounded delay and bounded clock drift models.Appendix Dincludes more models and requirements. The *same* models are used for PKI schemes in [25].

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 bounded clock drift. Lastly, we show that by additionally assuming bounded-delay communication, 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 security 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. Due to its modularity, MoSS also supports concrete security in a way we consider simple and even elegant; see Section 7.2.

Ensuring polytime interactions. As pointed out in [11, 22], the 'classical' notion of PPT algorithms is not sufficient for analysis of interactive 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 [21]. The extendability of MoSS allows it to handle these aspects relatively simply; (see Section 7.3 and Appendix B).

Modularity lemmas. In Section 5, we present several *asymptotic security modularity lemmas*, which allow combining 'simple' models and requirements into composite models and requirements, taking 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 [2,3,9,29], and the lack of *composability*, an important property proven for most simulation-based frameworks, most notably UC [11].

We believe that MoSS is amenable to computer-aided tools. For example, a tool may transform the modular MoSS security 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 security specifications, or to provide (MoSS-like) modular specifications for UC and other simulation-based frameworks.

It is our hope that MoSS may help to bridge the gap between the theory and practice in cryptography, and to 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 [14], but there are many other proposals, most notably, Certificate Transparency (CT) [24], which add significant goals and cryptographic mechanisms. Realistic PKI systems have non-trivial requirements; in particular, synchronization is highly relevant and needed to deal with even such basic aspects as revocation.

Recently, we presented the first rigorous study [25] of practical[4] PKI schemes by using MoSS. Specifically, we defined model and requirement predicates for practical PKI schemes and proved security of the X.509 PKI scheme. The analysis uses the bounded-delay and bounded-drift model predicates; similarly, follow-up work is expected to reuse these models and requirement predicates to prove security for additional PKI schemes, e.g., Certificate Transparency.

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

2 Execution Process

A key aspect of MoSS is the separation of the execution process from the model $\mathcal{M}$ under which a protocol $\mathcal{P}$ is analyzed, and the requirements $\mathcal{R}$ that define $\mathcal{P}$'s goals. 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 MoSS's 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.

[4] Grossly-simplified PKI ideal functionalities were studied, e.g., in [21], but without considering even basic aspects such as revocation and expiration.

2.1 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 a 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*. 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.

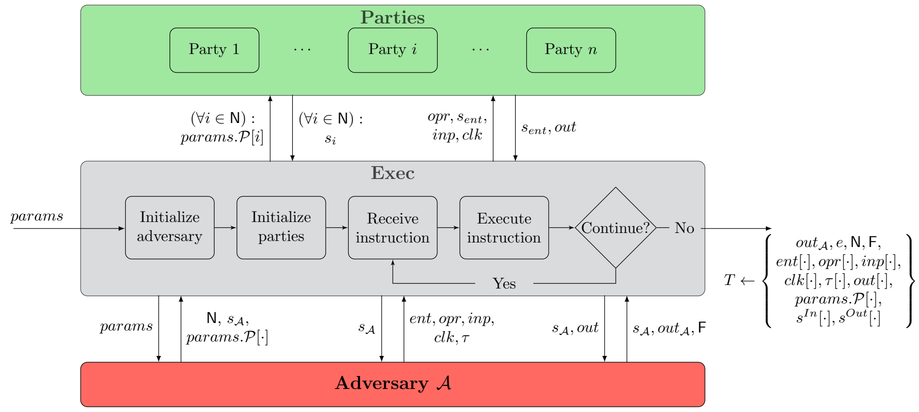
Parties
Party 1
· · ·
Party i
· · ·
Party n
$(\forall i \in \mathsf{N})$: $params.\mathcal{P}[i]$
$(\forall i \in \mathsf{N})$: s_i
opr, s_{ent}, inp, clk
s_{ent}, out
Exec
$params$
Initialize adversary
Initialize parties
Receive instruction
Execute instruction
Continue?
No
Yes
$T \leftarrow \left\{ out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], params.\mathcal{P}[\cdot], s^{In}[\cdot], s^{Out}[\cdot] \right\}$
$params$
$\mathsf{N}, s_{\mathcal{A}}, params.\mathcal{P}[\cdot]$
$s_{\mathcal{A}}$
ent, opr, inp, clk, τ
$s_{\mathcal{A}}, out$
$s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}$
Adversary $\mathcal{A}$

Fig. 2: A high level overview of MoSS's execution process showing the interactions between the parties to the protocol and the adversary in $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}$. (Note: e, in the final execution transcript T, is the total number of iterations of the loop.)

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 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.

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

```
1: $(s_\mathcal{A}, \mathsf{N}, params.\mathcal{P}[\cdot]) \leftarrow \mathcal{A}[\text{`Init'}](params)$    ▷ Initialize $\mathcal{A}$ with params
2: $\forall i \in \mathsf{N}:\ s_i \leftarrow \mathcal{P}[\text{`Init'}]\,(\bot, params.\mathcal{P}[i], \bot)$    ▷ Initialize entities' local states
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:     $s^{In}[e] \leftarrow s_{ent[e]}$    ▷ Save input state
8:     $(s_{ent[e]}, out[e]) \leftarrow \mathcal{P}\,[opr[e]]\,(s_{ent[e]}, inp[e], clk[e])$
9:     $s^{Out}[e] \leftarrow s_{ent[e]}$    ▷ Save output state
10:    $(s_\mathcal{A}, out_\mathcal{A}, \mathsf{F}) \leftarrow \mathcal{A}\,(s_\mathcal{A}, out[e])$    ▷ $\mathcal{A}$ decides when to terminate the loop ($out_\mathcal{A} \neq \bot$), based on $out[e]$
11: until $out_\mathcal{A} \neq \bot$
12: $T \leftarrow \Big(out_\mathcal{A}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], params.\mathcal{P}[\cdot], s^{In}[\cdot], s^{Out}[\cdot]\Big)$
13: Return $T$    ▷ Output transcript of run
```

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

Initialization (lines 1-3). In line 1, we allow the adversary to set their state $s_{\mathcal{A}}$, to choose the set of entities N, and to choose parameters $params.\mathcal{P}[i]$ for protocol initialization for each entity $i \in \mathsf{N}$. Note that each $params.\mathcal{P}[i]$ can include all of the parameters *params* given to the execution process, some of the parameters from *params*, or entirely different parameters, chosen by the adversary; however, the allowed values of $params.\mathcal{P}[\cdot]$ (including their relation to *params*) can be restricted using *models* (see Sec. 3), since the values set by the adversary are returned in the transcript T (lines 12-13). In line 2, we set the initial state s_i for each entity i by invoking the protocol-specific 'Init' operation with input $params.\mathcal{P}[i]$; 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 4-11), 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 4-11).

Main execution loop (lines 4-11). 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 8). The entity's input and output states are saved in $s^{In}[e]$ and $s^{Out}[e]$, respectively (lines 7 and 9), which allows models to place restrictions on the states of entities.

In line 10, 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 $out_{\mathcal{A}} \neq \bot$, the execution moves to the termination phase; otherwise the loop continues.

Termination (lines 12-13). Upon termination, the process returns the *execution transcript* T (line 13), 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 10), the values of $ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot]$ and $out[\cdot]$ for all invoked events, the protocol initialization parameters $params.\mathcal{P}[\cdot]$ for all entities in N, and the entity's input state $s^{In}[\cdot]$ and output state $s^{Out}[\cdot]$ for each event. 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) and (2) define certain other models/requirements, e.g., indistinguishability requirements (Section 4.3); 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[5] $\mathcal{X}$ 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 an entity, 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. 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 $\textit{sec-out}[e][\cdot]$ 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.

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' $s_{\mathcal{X}}$; this state is used by execution operations (in $\mathcal{X}$), allowing them to be defined as (stateless) functions. Note that 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 $(params, params.\mathcal{P}[\cdot])$. The rest of the initialization lines are the same.

The main execution loop (lines 5-16) is as before, but with one difference, where the adversary $\mathcal{A}$ determines in line 7 the type of operation $type[e]$ to be invoked by an entity $ent[e] \in \mathsf{N}$. The operation type $type[e] \in \{\text{`}\mathcal{X}\text{'}, \text{`}\mathcal{P}\text{'}\}$

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

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 9-12) 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 17-18) 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: 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 $\mathcal{X}$). Also, in lines 7 and 15, the operation to $\mathcal{A}$ is not explicitly written in the pseudo-code. 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 *specification operations* and *entity-fault 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. C.2. We define the 'Sec-in' operation in Equation 1.[6]

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

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

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

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

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

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

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

'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.1.2, 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[7] to define both models (in this section) and requirements (in Sec. 4).

A MoSS (model/requirement) specification is a pair of functions (π, β), where $\pi(T, params)$ is called the *predicate* (and returns $\top$ or $\bot$) and $\beta(params)$ is the *base (probability) function* (and evaluates to values from 0 to 1). 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*.

A MoSS model is 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.: $\pi(T, params) = \top$, 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'*.

[7] We use the term 'specification' to refer to a *component* of a model (or of a requirement - see Sec. 4). This is not to be confused with 'security specification', which we use to mean a model, requirement, and specific execution process.

The separation between the execution process and the model allows to use the same - relatively simple - execution process for the analysis of many different protocols, under different models (of the environment and adversary capabilities). Furthermore, it allows 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*[8] of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π using execution operations $\mathcal{X}$, as a function of the parameters $params$. This is the probability that $\pi(T, params) = \bot$, for the transcript T of a random execution: $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$.

[8] Note that the advantage of $\mathcal{A}$ is the *total* probability of $\mathcal{A}$ winning, i.e., it does not depend on a base function.

Definition 1 (Advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π using execution operations $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X}$ be algorithms 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:*

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

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 *poly-satisfy model $\mathcal{M}$ using execution operations $\mathcal{X}$*.

Definition 2 (Adversary $\mathcal{A}$ poly-satisfies model $\mathcal{M}$ 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 $\mathcal{A}$ poly-satisfies model $\mathcal{M}$ 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.:*

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

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

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

Algorithm 3 The $\pi^{\text{Drift}}_{\Delta_{clk}}$ $(T, params)$ predicate, used by the $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}} \equiv (\pi^{\text{Drift}}_{\Delta_{clk}}, 0)$ model

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$:                          ▷ For each event
3:         $|T.clk[\hat{e}] - T.\tau[\hat{e}]| \leq \Delta_{clk}$         ▷ Local clock is within $\Delta_{clk}$ 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 ($T = \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$) and $params$ are the parameters, i.e., $b \leftarrow \pi(T, params)$; if $b = \perp$ then we say that *requirement predicate* π *was not satisfied* in the execution of $\mathcal{P}$, or that the *adversary won* in this execution. If $b = \top$, then we say that *requirement predicate* π *was satisfied* in this execution, or that the *adversary lost.*

We now define what it means for $\mathcal{P}$ *to poly-satisfy* $\mathcal{R}$ *under model* $\mathcal{M}$ *using execution operations* $\mathcal{X}$.

Definition 3 (Protocol $\mathcal{P}$ poly-satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$). *Let $\mathcal{P}, \mathcal{X} \in PPT$, and let $\mathcal{R}$ be a set of specifications, i.e., $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$. We say that protocol $\mathcal{P}$ poly-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 poly-satisfies $\mathcal{M}$ using execution operations $\mathcal{X}$, every parameters params $\in \{0,1\}^*$, and 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.:*

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

4.2 Example: the No False Accusations Requirement $\mathcal{R}_{\mathsf{NFA}}$

Intuitively, the *No False Accusations (NFA)* requirement $\mathcal{R}_{\mathsf{NFA}}$ states that a non-faulty entity $a \notin \mathsf{F}$ would *never (falsely) accuse* of a fault another non-faulty entity, $b \notin \mathsf{F}$. It is defined as $\mathcal{R}_{\mathsf{NFA}} = (\pi_{\mathsf{NFA}}, 0)$. To properly define the π_{NFA} requirement predicate, we first define a convention for one party, say $a \in \mathsf{N}$, to output an Indicator of Accusation, i.e., 'accuse' another party, say $i_{\mathrm{M}} \in \mathsf{N}$, 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}, 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 $ent[\hat{e}_A]$ accused i_{M}; 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., $j, T.ent[e] \in T.\mathsf{N} - T.\mathsf{F}$).

Algorithm 4 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
   )
```

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'.

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

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

  where $inp[e].opr \in \mathcal{P}$ (one of the operations in $\mathcal{P}$) and $inp[e].inp$ is an 'array' with two possible inputs, of which only one is randomly chosen via $s_{\mathcal{X}}.b$, hence, the $inp[e].inp[s_{\mathcal{X}}.b]$ notation.
- 'Guess': checks if a 'guess bit', which is provided by the adversary as input, is equal to $s_{\mathcal{X}}.b$, and returns the result in $sec\text{-}out[e]$. 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 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 $s_{\mathcal{X}}.b$, $\mathcal{A}$ neither has any knowledge about which input is selected nor 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 The Generic Indistinguishability Requirement $\mathcal{R}^{\pi}_{\text{IND}}$ and the Message Confidentiality Requirement $\mathcal{R}^{\pi_{\text{MsgConf}}}_{\text{IND}}$

To illustrate how the aforementioned operations can be used in practice, we define the indistinguishability requirement $\mathcal{R}^{\pi}_{\text{IND}}$ as $\mathcal{R}^{\pi}_{\text{IND}} = (\text{IND}^{\pi}, \frac{1}{2})$, where the IND^{π} predicate is shown 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. The base function of the $\mathcal{R}^{\pi}_{\text{IND}}$ requirement is $\frac{1}{2}$, because the probability that the adversary guesses correctly should not be significantly more than $\frac{1}{2}$.

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

```
1: return ¬(
2:     $T.type[T.e] = \text{‘}\mathcal{X}\text{’}$
3:     and $T.opr[T.e] = \text{‘Guess’}$ 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 requirements; for example, we use the π_{MsgConf} predicate (Algorithm 6) to define $\mathcal{R}_{\text{IND}}^{\pi_{\mathsf{MsgConf}}} = (\text{IND}^{\pi_{\mathsf{MsgConf}}}, \frac{1}{2})$, which defines 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 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 at the challenge receiver receiving from the challenge sender 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. $T.type[\hat{e}] = \text{‘}\mathcal{X}\text{’}$ and $T.opr[\hat{e}] = \text{‘Challenge’}$:
3:         $T.inp[\hat{e}].opr = \text{‘Send’}$    ▷ Every ‘Challenge’ event is for ‘Send’ operation
4:         and $|T.inp[\hat{e}].inp[0].m| = |T.inp[\hat{e}].inp[1].m|$    ▷ Messages have equal length
5:         and $\exists\ i_{\text{S}}, i_{\text{R}} \in T.\mathsf{N}$ s.t.    ▷ There is one specific sender $i_S$ and one specific receiver $i_R$
6:             $T.inp[\hat{e}].inp[0].i_{\text{R}} = T.inp[\hat{e}].inp[1].i_{\text{R}} = i_{\text{R}}$    ▷ $i_R$ is the recipient for both messages
7:             and $T.ent[\hat{e}] = i_{\text{S}}$    ▷ $i_S$ is the sender
8:             and $\nexists\ \hat{e}'$ s.t. $T.opr[\hat{e}'] = \text{‘Receive’}$    ▷ There is no ‘Receive’ event $\hat{e}'$
                and $T.inp[\hat{e}'].c = T.out[\hat{e}].c$
9:              and $T.ent[\hat{e}'] = i_{\text{R}}$    ▷ Where $\mathcal{A}$ uses decrypts the output of the challenge
                and $T.inp[\hat{e}'].i_{\text{S}} = i_{\text{S}}$
   )
```

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 $\mathcal{M}$ as *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 the *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 $\mathcal{A}$ poly-satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$, then $\mathcal{A}$ poly-satisfies $\mathcal{M}$ 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)*, 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 $\mathcal{A}$ poly-satisfies both $\mathcal{M}$ and $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{A}$ poly-satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ 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)*, 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 $\mathcal{P}$ poly-satisfies requirement $\mathcal{R}$ under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies $\mathcal{R}$ 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)* 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 $\mathcal{P}$ poly-satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies $\mathcal{R}$ 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)* 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 $\mathcal{P}$ poly-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and poly-satisfies $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies 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 $\mathcal{X}$, namely:

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

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 [25], 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 C, 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 a few details as examples of the use of MoSS; in particular, AuthBroadcast addresses shared-key initialization, an aspect which does not exist in PKI schemes. We define $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{'Sec-in'}]}$ and $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{'Sec-in'}]}$, two simple models for shared-key initialization. These models 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 a requirement - assuming increasingly stronger models. Note that in the analysis, we also assume the $\mathcal{M}_{\text{polyAdv}}$ model, defined in Sec 7.3, in order to only consider adversaries whose total runtime during the execution is bounded by a polynomial; this is omitted below for simplicity.

1. We first show that AuthBroadcast ensures *authentication* of received messages assuming that a key is shared securely once among all entities and valid n and 1^κ parameters are given to the protocol. Namely, we show that AuthBroadcast poly-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}_{\text{SecKeyInit}}$ using $\mathcal{X}$-operations {'Sec-in'}.

2. We then show that AuthBroadcast ensures *authentication and freshness* of received messages under a stronger model that also assumes a weak-level of clock synchronization (bounded clock drift). Namely, we show that AuthBroadcast poly-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$-operations {'Sec-in'} for $f(\Delta) = \Delta + 2\Delta_{clk}$, where Δ_{clk} is the assumed maximal clock drift.
3. Finally, we show that AuthBroadcast ensures *correct bounded-delay delivery/receipt* of broadcast messages (which implies authenticity and freshness as well) under an even stronger model which also assumes a bounded delay of communication and a sufficiently large freshness interval given to the protocol. Specifically, we show that AuthBroadcast poly-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ using $\mathcal{X}$-operations {'Sec-in'}, where Δ_{clk} is the assumed maximal clock drift and Δ_{com} is the assumed maximal communication delay.

6.2 Specifications for PKI Scheme

PKI schemes are an essential building block for protocols utilizing public key cryptography. Unfortunately, there have been multiple incidents and vulnerabilities involving PKI, resulting in extensive research on improving security of PKI. Provably-secure PKI schemes were presented in [13], however, these specifications did not cover aspects critical in practice, such as timely revocation or transparency. We next briefly discuss one of the PKI security specifications defined using MoSS.

Sample model: $\mathcal{M}^{Drift}_{\Delta_{clk}}$. [25] defines several models covering assumptions regarding the adversary capabilities, the environment (communication and synchronization) and the initialization, assumed by different PKI protocols. The bounded clock drift model $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ (presented in Section 3.3) is an example of a *generic model* which is common to many applied protocols and can be reused among different works and tasks.

Sample requirement: ΔTRA. PKI schemes have multiple security requirements, from simple requirements such as accountability to more complex requirements such as equivocation detection and prevention as well as transparency. Intuitively, the Δ-transparency (ΔTRA) requirement specifies that a certificate attested as Δ-transparent must be available to all 'interested' parties, i.e., *monitors*, within Δ time of its transparency attestation being issued by a proper authority, typically referred to as a *logger*. This requirement is defined as the pair $(\pi_{\Delta\mathsf{TRA}}, 0)$, where the $\pi_{\Delta\mathsf{TRA}}$ predicate is defined in Algorithm 7, as a conjunction of the simple sub-predicates, defined in [25].

Let us explain the operation of $\pi_{\Delta\mathsf{TRA}}$. This predicate ensures that for a certificate ψ and Δ-transparency attestation ρ as attested by an entity $\rho.\iota$, there is an honest entity $\iota \in \mathsf{N}$ (HonestEntity), and ι confirmed that $\rho.\iota$'s public key is pk (CorrectPublicKey). Then, it verifies that ψ is a valid certificate attested as Δ-transparent using ρ (ValidCertificateAttestation). *However*, there exists another *honest* entity $\iota_M \in \mathsf{N}$ (HonestEntity) which monitors $\rho.\iota$ (IsMonitor) but is unaware of ψ (HonestMonitorUnawareOfCertificate)

Algorithm 7 The Δ-transparency ($\Delta\mathsf{TRA}$) predicate $\pi_{\Delta\mathsf{TRA}}$

$$\pi_{\Delta\mathsf{TRA}}(T, params) \equiv \left\{ \begin{array}{l} (\psi, \rho, pk, \iota, \iota_M) \leftarrow T.out_{\mathcal{A}}; \\ \textbf{return } \neg \left[\begin{array}{l} \textsc{HonestEntity}(T, params, \iota) \wedge \\ \textsc{CorrectPublicKey}(T, params, \iota, pk, \rho.\iota) \wedge \\ \textsc{ValidCertificateAttestation}(T, params, \{\Delta\mathsf{TRA}\}, \psi, pk, \rho) \wedge \\ \textsc{HonestEntity}(T, params, \iota_M) \wedge \\ \textsc{IsMonitor}(T, params, \iota_M, \rho.\iota) \wedge \\ \textsc{HonestMonitorUnawareOfCertificate}(T, params, \psi, \rho) \wedge \\ \textsc{WasNotAccused}(T, params, \iota_M, \rho.\iota) \end{array} \right]; \end{array} \right\}$$

- although it should, and yet, there was no accusation of misbehavior issued[9] (WasNotAccused).

This design for a predicate as a conjuncture of sub-predicate is typical and rather intuitive, and it illustrates another aspect of modularity: the sub-predicates are easy to understand and validate, and are also reusable; for example, a predicate to validate an entity's public key (ValidCertificateAttestation) or that an entity is honest (HonestEntity) can be useful for other, unrelated to PKI protocols.

7 Concrete Security and Ensuring Polytime Interactions

In this section, we present the CS *compiler* (Sec. 7.1), which transforms the adversary into an 'equivalent' algorithm, which provides three additional outputs: the total runtime of the adversary, the number of bit flips by the adversary, and the initial size of the adversary's state. We then use the CS compiler for two applications. First, in Sec. 7.2, we extend MoSS to support *concrete security.* Finally, in Sec. 7.3, we show how the CS compiler allows to ensure *polytime interactions*, and in particular, limit the adversary so that its runtime is polynomial in the security parameter.

7.1 The CS Compiler

The extension that will allow us to give concrete security definitions (Sec. 7.2) and to enforce polytime interactions (Sec. 7.3), is a *compiler*, denoted CS (which stands for both '*CtrSteps*' and 'Concrete Security').

The input to CS is an (adversary) algorithm $\mathcal{A}$, and the output, $\mathsf{CS}(\mathcal{A})$, is an algorithm which outputs the same output as $\mathcal{A}$ would produce, and three additional values, added to the final $out_{\mathcal{A}}$ output of $\mathcal{A}$: $out_{\mathcal{A}}.CtrSteps$, the number of steps of $\mathcal{A}$ throughout the execution; $out_{\mathcal{A}}.CtrBitFlips$, the number of bit-flip operations performed by $\mathcal{A}$; and $out_{\mathcal{A}}.LenInitState$, the size of the initial state output by $\mathcal{A}$.

[9] Notice that ι, ι_M are honest, but $\rho.\iota$ is not necessarily honest, and therefore, WasNotAccused is needed, because $\rho.\iota$ might not cooperate in order for ι_M to not be aware of ψ.

Now, instead of running the execution process directly over input adversary $\mathcal{A}$, we run $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$, i.e., we run the 'instrumented' adversary $\mathsf{CS}(\mathcal{A})$. This way, in the execution transcript, we receive these three measured values ($out_{\mathcal{A}}.CtrSteps$, $out_{\mathcal{A}}.CtrBitFlips$ and $out_{\mathcal{A}}.LenInitState$). It remains to describe the operation of CS.

Note that CS maintains its own state, which contains, as part of it, the state of the adversary $\mathcal{A}$. This creates a somewhat confusing situation, which may be familiar to the reader from constructions in the theory of complexity, or, esp. to practitioners, from the relation between a virtual machine and the program it is running. Namely, the execution process received the algorithm $\mathsf{CS}(\mathcal{A})$ as the adversary, while $\mathsf{CS}(\mathcal{A})$ is running the 'real' adversary $\mathcal{A}$. Thus, the state maintained by the execution process is now of $\mathsf{CS}(\mathcal{A})$; hence, we refer to this state as $s_{\mathsf{CS}(\mathcal{A})}$.

The state $s_{\mathsf{CS}(\mathcal{A})}$ consists of four variables. The first variable contains the state of the original adversary $\mathcal{A}$. We denote this variable by $s_{\mathsf{CS}(\mathcal{A})}.s_{\mathcal{A}}$; 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 $s_{\mathsf{CS}(\mathcal{A})}$ of the $\mathsf{CS}(\mathcal{A})$ algorithm (run as the adversary by the execution process).

The other three variables in the state $s_{\mathsf{CS}(\mathcal{A})}$ are invisible to $\mathcal{A}$, since they are not part of $s_{\mathsf{CS}(\mathcal{A})}.s_{\mathcal{A}}$. These are: $s_{\mathsf{CS}(\mathcal{A})}.CtrSteps$, a counter which the algorithm $\mathsf{CS}(\mathcal{A})$ uses to sum up the total runtime (steps) of $\mathcal{A}$; $s_{\mathsf{CS}(\mathcal{A})}.CtrBitFlips$, a counter which $\mathsf{CS}(\mathcal{A})$ uses to sum up the number of random bits flipped by $\mathcal{A}$; and, finally, $s_{\mathsf{CS}(\mathcal{A})}.LenInitState$, which stores the size of the initial state output by $\mathcal{A}$.

Whenever the execution process invokes $\mathsf{CS}(\mathcal{A})$, then $\mathsf{CS}(\mathcal{A})$ 'runs' $\mathcal{A}$ on the provided inputs, measuring the time (number of steps) until $\mathcal{A}$ returns its response, as well as the number of random bits (coin flips) used by $\mathcal{A}$. When $\mathcal{A}$ returns a response, $\mathsf{CS}(\mathcal{A})$ increments the $s_{\mathsf{CS}(\mathcal{A})}.CtrSteps$ counter by the run-time of $\mathcal{A}$ in this specific invocation and increments the $s_{\mathsf{CS}(\mathcal{A})}.CtrBitFlips$ counter by the number of bit flips of $\mathcal{A}$ in this invocation. When $\mathcal{A}$ returns a response $(s_{\mathcal{A}}, \mathsf{N}, params.\mathcal{P}[\cdot])$ after being invoked by $\mathsf{CS}(\mathcal{A})[\text{'Init'}](params)$ in line 1, then $\mathsf{CS}(\mathcal{A})$ additionally sets $s_{\mathsf{CS}(\mathcal{A})}.LenInitState \leftarrow |s_{\mathcal{A}}|$. Finally, $\mathsf{CS}(\mathcal{A})$ checks if $\mathcal{A}$ signaled termination of the execution process. When $\mathcal{A}$ signals termination (by returning $out_{\mathcal{A}} \neq \bot$), then the $\mathsf{CS}(\mathcal{A})$ algorithm sets $out_{\mathcal{A}}.CtrSteps$, $out_{\mathcal{A}}.CtrBitFlips$, and $out_{\mathcal{A}}.LenInitState$ to $s_{\mathsf{CS}(\mathcal{A})}.CtrSteps$, $s_{\mathsf{CS}(\mathcal{A})}.CtrBitFlips$, and $s_{\mathsf{CS}(\mathcal{A})}.LenInitState$, respectively, i.e., adds to $out_{\mathcal{A}}$ the computed total runtime of $\mathcal{A}$ during this execution, the number of bit flips of $\mathcal{A}$ during this execution, and the size of the initial state output by $\mathcal{A}$[10]; of course, we still have $out_{\mathcal{A}} \neq \bot$ and therefore the execution process terminates - returning as part of $out_{\mathcal{A}}$ the total runtime of $\mathcal{A}$ and the size of the initial state output by $\mathcal{A}$. Although these values are carried in $out_{\mathcal{A}}$, the adversary cannot modify or view them.

[10] Note this would override any values that $\mathcal{A}$ may write on $out_{\mathcal{A}}.CtrSteps$, $out_{\mathcal{A}}.CtrBitFlips$, and $out_{\mathcal{A}}.LenInitState$, i.e., we essentially forbid the use of $out_{\mathcal{A}}.CtrSteps$, $out_{\mathcal{A}}.CtrBitFlips$, and $out_{\mathcal{A}}.LenInitState$ by $\mathcal{A}$.

7.2 Concrete Security

We new describe how we can use CS to support *concrete security* [6] in MoSS. In concrete security, the adversary's advantage is a function of 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'). Notice that since we explicitly bound the adversary's runtime, we do not need to require the adversary to be a PPT algorithm.

To be more specific, we provide *bounds* on adversary resources, including runtime and number of coin-flips (random bits), as *parameters* in *params*; this allows the adversary to limit its use of resources accordingly. We (next) define the *Concrete Security model* $\mathcal{M}^{CS}$, which validates that the adversary, indeed, does not exceed the bounds specified in *params*. To validate the bounds on the adversary's runtime and number of coin-flips (random bits), $\mathcal{M}^{\text{CS}}$ uses $out_{\mathcal{A}}.CtrSteps$ and $out_{\mathcal{A}}.CtrBitFlips$, hence, this model should be applied to the transcript $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$, produced by running the 'instrumented adversary' $\mathsf{CS}(\mathcal{A})$.

7.2.1 The Concrete Security Model $\mathcal{M}^{\text{CS}}$ and Resource Bounds

Concrete security defines the adversary's advantage as a function of the *bounds* on adversary resources, specified in *params*. Specifically, we adopt the following conventions for the adversary resource parameters. First, *params* includes an array $params.bounds.maxCalls$, where each entry $params.bounds.maxCalls[type][opr]$ contains the maximum number of calls that $\mathcal{A}$ is allowed to make to operation *opr* of type *type*. Second, *params* includes the field $params.bounds.maxSteps$, which is the maximum number of steps that the adversary is allowed to take, and the field $params.bounds.maxBitFlips$, which is the maximum number of bit flips that the adversary is allowed to use.

The Concrete Security model $\mathcal{M}^{\text{CS}}$ validates that the adversary never exceeds these bounds; it is defined as $\mathcal{M}^{\text{CS}} = \left\{(\pi^{\text{CS}}, 0)\right\}$, i.e., we expect the adversary to *always* limit itself to the bounds specified in $params.bounds$.

The π^{CS} predicate (Algorithm 8) ensures that: (1) $\mathcal{A}$ does not exceed the bounds in $params.bounds.maxCalls$ on the number of calls to each operation, (2) $\mathcal{A}$ does not exceed the bound $params.bounds.maxSteps$ on the number of steps it takes, and (3) $\mathcal{A}$ does not exceed the bound $params.bounds.maxBitFlips$ on the number of bit flips it uses.

7.2.2 Satisfaction of Concrete-Security Models and Requirements

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

Algorithm 8 $\pi^{\text{CS}}(T, params)$ Predicate

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

β. Similarly, when $\mathcal{P}$ satisfies $\mathcal{R}$ under some model $\mathcal{M}$, for every $\mathcal{A}$ that satisfies $\mathcal{M}$ and every specification in $\mathcal{R}$, the probability of $\mathcal{A}$ winning is *bounded* by the base function β.

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 function that returns values in $[0, 1]$, which depend on the resources available to the adversary, e.g., maximal runtime (steps). This difference should be familiar to readers familiar with concrete-security definitions and results, e.g., [5]. However, often we can use the same predicate π in both types of analysis.

We now give the concrete definition of a model-satisfying adversary. Note that the base function $\beta(params)$ is a function of the parameters $(params)$, including the bounds on the adversary resources $(params.bounds)$. To make these bounds meaningful, a model-satisfying adversary *always* has to satisfy $\mathcal{M}^{\text{CS}}$ (see § 7.2.1).

Definition 4 (Adversary $\mathcal{A}$ CS-satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{X}$ be algorithms and let $\mathcal{M}$ be a set of specifications, i.e., $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$. We say that adversary $\mathcal{A}$ CS-satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$, denoted $\mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \mathcal{M}$, if for every protocol $\mathcal{P}$, $params \in \{0,1\}^*$, and specification $(\pi, \beta) \in \mathcal{M} \cup \mathcal{M}^{CS}$, the advantage of $\mathsf{CS}(\mathcal{A})$ against $\mathcal{P}$ for π using $\mathcal{X}$ is bounded by $\beta(params)$, i.e.:*

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \mathcal{M} \overset{def}{=} \left[\begin{array}{c} \left(\forall\ \mathcal{P}, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M} \cup \mathcal{M}^{CS}\right) : \\ \epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(params) \leq \beta(params) \end{array} \right] \tag{10}$$

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

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

$$\mathcal{P} \models_{\mathsf{CS}}^{\mathcal{M},\,\mathcal{X}} \mathcal{R} \overset{def}{=} \begin{bmatrix} (\forall\ \mathcal{A}\ s.t.\ \mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \mathcal{M},\ params \in \{0,1\}^*,\ (\pi,\beta) \in \mathcal{R}) : \\ \epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(params) \leq \beta(params) \end{bmatrix} \tag{11}$$

Note that if adversary $\mathcal{A}$ CS-satisfies $\mathcal{M}$ using $\mathcal{X}$ for a model $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$ where every base function is a positive negligible function in the security parameter (i.e., $|params|$), then $\mathcal{A}$ poly-satisfies $\mathcal{M}'$ using $\mathcal{X}$ for $\mathcal{M}' = \{(\pi_1, 0), \ldots\}$ - i.e., $\mathcal{A}$ satisfies a model with the same predicates as $\mathcal{M}$ but with all zero-constant base functions in the asymptotic sense. Similarly, if protocol $\mathcal{P}$ CS-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ for a requirement $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$ where every base function is a positive negligible function in $|params|$, then $\mathcal{P}$ poly-satisfies $\mathcal{R}'$ under $\mathcal{M}$ using $\mathcal{X}$ for $\mathcal{R}' = \{(\pi_1, 0), \ldots\}$.

7.3 Ensuring Polytime Interactions

We next discuss a very different application of the CS Compiler (subsection 7.1): ensuring polytime interactions. Let us first explain the polytime interaction challenge. In most of this work, as in most works in cryptography, we focus on PPT algorithms and asymptotically polynomial specifications. For instance, consider Definition 2, where we require $\mathcal{A}, \mathcal{X}, \mathcal{P} \in PPT$ and bound the advantage by the base function plus a negligible function - i.e., a function which is smaller than any positive polynomial in the length of the inputs, for sufficiently large inputs.

However, when analyzing interacting systems as facilitated by MoSS, there is a concern: each of the algorithms might be in PPT, yet the *total* runtime can be *exponential* in the size of the *original input*. For example, consider an adversary $\mathcal{A}$, that, in every call, outputs a state which is twice the size of its input state. Namely, if the size of the adversary's state in the beginning was l, then after e calls to the adversary algorithm $\mathcal{A}$, the size of $s_{\mathcal{A}}$ would be $2^e \cdot l$, i.e., exponential in the number of steps e.

For asymptotic analysis, we may want to ensure *polytime interactions*, i.e., to limit the total running time of $\mathcal{A}$ and $\mathcal{P}$ during the execution to be polynomial. Let us first focus on the adversary's runtime. To limit the adversary's total runtime by a polynomial in the length of its initial input, i.e., length of *params*, we use the CS Compiler, i.e., consider the execution transcript of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$. Specifically, we use the fact that the transcript T includes the size of the initial state output by $\mathcal{A}$ in $T.s_{\mathcal{A}}.LenInitState$, as well as the total number of steps taken by $\mathcal{A}$ in $T.s_{\mathcal{A}}.CtrSteps$.

Define the model $\mathcal{M}_{\text{polyAdv}}$ as $\mathcal{M}_{\text{polyAdv}} = (\pi_{\text{polyAdv}}, 0)$, where the π_{polyAdv} predicate, shown in Algorithm 9, verifies that $T.s_{\mathcal{A}}.CtrSteps$ is bounded by $2 \cdot T.s_{\mathcal{A}}.LenInitState$. When T is a transcript returned by $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$, this means that the number of steps taken by $\mathcal{A}$ over the whole execution does not exceed twice[11] the size of the initial state output by $\mathcal{A}$, which is bounded by a polynomial in $|params|$. Hence, model $\mathcal{M}_{\text{polyAdv}}$ ensures that the total runtime of the adversary, over the entire execution, is polynomial in the size of the input parameters.

Algorithm 9 The π_{polyAdv} $(T,\, params)$ Predicate

```
1: return $(T.out_{\mathcal{A}}.CtrSteps \leq 2 \cdot T.out_{\mathcal{A}}.LenInitState)$
```

The $\mathcal{M}_{\text{polyAdv}}$ model ensures polynomial runtime of the adversary, and hence also a polynomial number of invocations of the protocol. In some situations it is also important to similarly restrict the *protocols*, e.g., when proving an impossibility or lower-bound on protocols. Note that for most 'real' protocols, such restrictions hold immediately from assuming the protocol is a PPT algorithm, since such protocols use bounded-size state and messages (outputs); and if we assume polynomial adversaries, then the total runtime of such protocols is polynomial even if we allow linear growth in state and outputs. We can focus on 'reasonable' protocols by including an appropriate requirement in the specifications. More details are included in App. B.

8 Conclusions and Future Work

The MoSS framework enables modular security 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 models, and even some generic requirements, may be reused across different works. 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 specifically, possibly using the modularity lemmas of section 5, (2) extending the MoSS framework to support secure composition, and (3) exploring the ability to support MoSS-like modular specifications in simulation-based frameworks such as UC, and the ability to support simulation-based specifications in MoSS. Finally, we hope that MoSS will prove useful in specification and analysis of applied protocols, and the identification and reuse of standard and generic models and requirements.

[11] We allow the total runtime to be *twice* the length of the adversary's initial state, to give the adversary additional time so it can also output this initial state, and is left with enough time for the execution.

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. Special thanks to Oded Goldreich for his encouragement and for suggesting a simplified way to ensure total polynomial time, which was the basis for our current 'interactive polytime adversary' (Section 7.3). Part of the work was done while Ewa Syta had a visiting position at 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., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. J. Comput. Syst. Sci. **61**(3), 362–399 (2000)
7. 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)
8. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: EUROCRYPT (2006)
9. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Transactions on Dependable and Secure Computing (2008)
10. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible universal composability made simple. In: EUROCRYPT (2019)
11. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: IEEE Symposium on Foundations of Computer Science (2001)
12. Canetti, R.: Universally composable security. Journal of the ACM (JACM) **67**(5), 1–94 (2020)
13. 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`
14. CCITT, B.B.: Recommendations X. 509 and ISO 9594-8. Information Processing Systems-OSI-The Directory Authentication Framework (Geneva: CCITT) (1988)
15. Degabriele, J.P., Paterson, K., Watson, G.: Provable security in the real world. IEEE Security & Privacy **9**(3), 33–41 (2010)

16. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM **33**(4), 792–807 (Oct 1986)
17. Goldwasser, S., Micali, S.: Probabilistic Encryption. Journal of Computer and System Sciences **28**(2), 270–299 (1984)
18. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on computing (1988)
19. Herzberg, A., Leibowitz, H., Syta, E., Wrótniak, S.: Moss: Modular security specifications framework - full version. Cryptology ePrint Archive, Report 2020/1040 (2020), `https://eprint.iacr.org/2020/1040`
20. Herzberg, A., Yoffe, I.: The layered games framework for specifications and analysis of security protocols. IJACT **1**(2), 144–159 (2008), `https://www.researchgate.net/publication/220571819_The_layered_games_framework_for_specifications_and_analysis_of_security_protocols`
21. Hofheinz, D., Shoup, V.: GNUC: A new universal composability framework. Journal of Cryptology **28**(3), 423–508 (2015)
22. Hofheinz, D., Unruh, D., Müller-Quade, J.: Polynomial runtime and composability. Journal of Cryptology **26**(3), 375–441 (2013)
23. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology pp. 1–124 (2020)
24. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Jun 2013). https://doi.org/10.17487/RFC6962
25. Leibowitz, H., Herzberg, A., Syta, E.: Provable security for PKI schemes. Cryptology ePrint Archive, Report 2019/807 (2019), `https://eprint.iacr.org/2019/807`
26. Leibowitz, H., Piotrowska, A.M., Danezis, G., Herzberg, A.: No Right to Remain Silent: Isolating Malicious Mixes. In: USENIX Security 19 (2019)
27. Lochbihler, A., Sefidgar, S.R., Basin, D., Maurer, U.: Formalizing constructive cryptography using crypthol. In: Computer Security Foundations (2019)
28. Maurer, U.: Constructive cryptography–a new paradigm for security definitions and proofs. In: Workshop on Theory of Security and Applications (2011)
29. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Computer Aided Verification (2013)
30. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004)
31. Wikström, D.: Simplified universal composability framework. In: Theory of Cryptography Conference. Springer (2016)

A Illustration of the Execution Process

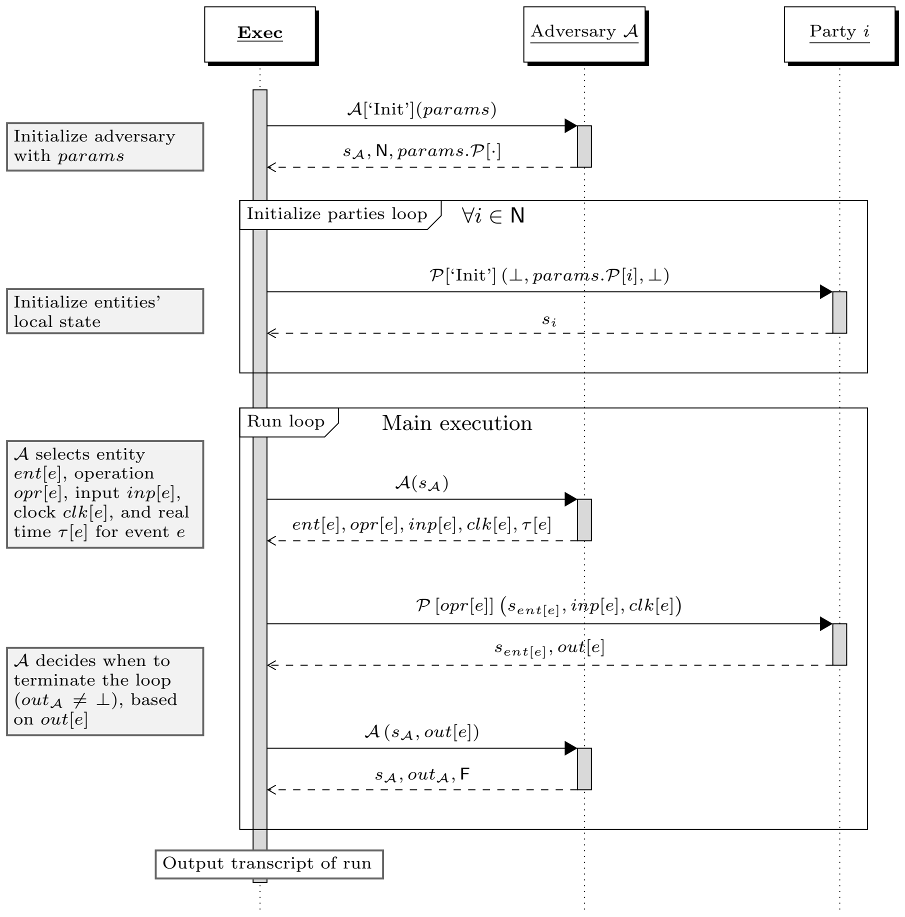
Exec
Adversary $\mathcal{A}$
Party i
$\mathcal{A}$['Init']($params$)
$s_\mathcal{A}, \mathsf{N}, params.\mathcal{P}[\cdot]$
Initialize adversary with $params$
Initialize parties loop
$\forall i \in \mathsf{N}$
$\mathcal{P}$['Init'] $(\perp, params.\mathcal{P}[i], \perp)$
s_i
Initialize entities' local state
Run loop
Main execution
$\mathcal{A}(s_\mathcal{A})$
$ent[e], opr[e], inp[e], clk[e], \tau[e]$
$\mathcal{A}$ selects entity $ent[e]$, operation $opr[e]$, input $inp[e]$, clock $clk[e]$, and real time $\tau[e]$ for event 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 execution process.

B Ensuring Polytime Interactions

In Sec. 7.3, we described how $\mathcal{A}$ and $\mathcal{P}$ may be PPT algorithms, but their runtime over the whole execution may not be polynomial (in $|params|$), which prevents reliance on (super-polynomial) hardness assumptions. To allow the use of asymptotic analysis and hardness assumptions, we may want to ensure *polytime interactions*, i.e., to limit the total running time of $\mathcal{A}$ and $\mathcal{P}$ during the execution to be polynomial.

This problem has been addressed in different ways, e.g., see two approaches in [11,22]. In Sec. 7.3, we showed a simple solution, i.e., how to limit the *adversary's* total runtime to be polynomial in the size of the parameters *params*, using the CS Compiler and the $\mathcal{M}_{\text{polyAdv}}$ model (Algorithm 9). This provides a simple way to ensure that the adversary's runtime is polynomial and that the adversary only gives polynomially-bounded inputs to the protocol. However, validating that an adversary satisfies $\mathcal{M}_{\text{polyAdv}}$ can be quite tricky and can require non-trivial analysis of its operation, and, moreover, there are some issues which the approach from Sec. 7.3 does not solve.

The first issue is that since the protocol also receives its *state* as input, directly from the execution process (not adversary), then a 'weird' protocol may output state whose length is, say, twice that of the input state, resulting in exponential state after a polynomial number of events. Such a protocol could thus have runtime which is exponential in $|params|$ - in total, and even in specific calls, after a linear number of steps.

A second possible problem is that a protocol may output strings which are too long for the adversary to handle. For example, consider a simple protocol that simply outputs messages twice the size of its input, and a model that requires the adversary to 'copy' these messages to another entity running the protocol. This would result in the output of the protocol growing exponentially (in the number of events), and soon enough, the adversary may run out of its polynomial runtime, which may prevent the adversary from 'winning' against a protocol, even if this protocol is clearly (intuitively) insecure.

These issues should not exist with realistic protocols, since their output size and state cannot grow and become exponential in the size of the original parameters. Thus, in this section, we present an alternative approach which addresses the above issues and ensures polynomial runtime (and outputs) of *both* the adversary and the protocol. Specifically, in Sec. B.1, we introduce two *compiler algorithms* ACom and PCom (similarly to the CS compiler in Sec. 7.1) which ensure that the outputs of the adversary and protocol are not too long, by *truncating the outputs*, if necessary, before outputting them to the execution process. In Sec. B.2, we give variants of the model-satisfying and requirement-satisfying definitions which use the two compilers. Note that in this approach, we assume that there are maximum input and output bounds for the protocol; the allowed values for these bounds can be restricted using models. Models can also be used to further restrict the outputs given to the protocol by the adversary in ways that depend on the protocol's maximum input and output bounds, as exemplified in Sec. B.3. In Sec. B.4, we discuss *bounded $\mathcal{X}$-operations*, which intuitively provide input and output limits similar to those provided by the compilers. Finally, in Sec. B.5, we prove the Polynomial Runtime Theorem, which states that in MoSS executions with bounded $\mathcal{X}$-operations, adversary $\mathsf{ACom}(\mathcal{A})$, and protocol $\mathsf{PCom}(\mathcal{P})$, for $\mathcal{A}, \mathcal{P} \in PPT$, the runtime of $\mathcal{A}$ and the runtime of $\mathcal{P}$ are polynomially-bounded (in $|params|$).

B.1 Adversary and Protocol Compilers

We start by defining two *compiler algorithms*, similarly to the CS compiler in Sec. 7.1: the ACom compiler for the adversary, and the PCom compiler for the protocol. In the next section, we will use these compilers to define variants of Def. 2 and 3, the model-satisfying and requirement-satisfying definitions. The purpose of the ACom compiler is to truncate outputs and output state of $\mathcal{A}$, so that they cannot get excessively large over the execution; to end the execution if the number of events gets too large; and to enforce some basic restrictions on three of the parameters given by the adversary to the protocol in $params.\mathcal{P}[\cdot]$ (specifically, the parameters 1^κ, $maxAOut$, and $maxPOut$). The purpose of the PCom compiler is to truncate outputs and output state of $\mathcal{P}$, to prevent them from growing excessively during the execution.

The input to ACom is an (adversary) algorithm $\mathcal{A}$ and the output is a modified adversary algorithm $\mathsf{ACom}(\mathcal{A})$. Pseudocode for the $\mathsf{ACom}(\mathcal{A})$ algorithm is shown in Algorithm 10. (Note that we use *trunc* to denote a truncation function - specifically, $trunc(m, x)$ returns the first x bits of m.) The algorithm $\mathsf{ACom}(\mathcal{A})$ runs $\mathcal{A}$ internally on provided inputs and then 'processes' its outputs as follows:

- After $\mathcal{A}$ is run for the first time (initialized) and outputs $params$, then $\mathsf{ACom}(\mathcal{A})$ overwrites some values in the protocol parameters. In particular, let j be the first entity output by $\mathcal{A}$ in N; then for all $i \in \mathsf{N}$, $\mathsf{ACom}(\mathcal{A})$ sets:
  - $params.\mathcal{P}[i].1^\kappa \leftarrow 1^{|params|}$
  - $params.\mathcal{P}[i].maxAOut \leftarrow params.\mathcal{P}[j].maxAOut$
  - $params.\mathcal{P}[i].maxPOut \leftarrow params.\mathcal{P}[j].maxPOut$

  It then outputs the modified $params$ instead of the $params$ output by $\mathcal{A}$. This ensures that the protocol receives the security parameter (that is, $1^{|params|}$) and the same maximum input and output sizes for all entities. Any other parameters chosen by $\mathcal{A}$ to give to the protocol are not altered.
- Every time after this, when $\mathcal{A}$ is run, if the output of $\mathcal{A}$ (not including its state) is longer than the protocol's maximum input size, then $\mathsf{ACom}(\mathcal{A})$ truncates it to have size $|params.maxAOut|$. Similarly, if $\mathcal{A}$ outputs a new state that is longer than its previous state, then $\mathsf{ACom}(\mathcal{A})$ truncates it to have the same size as the initial $|s_\mathcal{A}|$. The $\mathsf{ACom}(\mathcal{A})$ algorithm outputs the truncated outputs and state to the execution process.
- The $\mathsf{ACom}(\mathcal{A})$ algorithm keeps track of how many times it has been invoked to know how many events have happened. If the event number is already $|s_\mathcal{A}|$ then, in the second invocation of the adversary, $\mathsf{ACom}(\mathcal{A})$ outputs $out_\mathcal{A} = \top$ to end the execution.

Algorithm 10 $\mathsf{ACom}(\mathcal{A})[opr](inputs)$

```
1: if $opr$ = 'Init' then
2:     $params \leftarrow inputs$
3:     $(s_{\mathcal{A}}, \mathsf{N}, params.\mathcal{P}[\cdot]) \leftarrow \mathcal{A}[\text{'Init'}](params)$
4:     $s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}} \leftarrow s_{\mathcal{A}}$
5:     $s_{\mathsf{ACom}(\mathcal{A})}.numEvents \leftarrow 0$
6:     $s_{\mathsf{ACom}(\mathcal{A})}.maxStateSz \leftarrow |s_{\mathcal{A}}|$
7:     $s_{\mathsf{ACom}(\mathcal{A})}.maxAOut \leftarrow \max\{\{1\} \cup \{params.\mathcal{P}[i].maxAOut\}_{i \in \mathsf{N}}\}$
8:     $s_{\mathsf{ACom}(\mathcal{A})}.maxPOut \leftarrow \max\{params.\mathcal{P}[i].maxPOut\}_{i \in \mathsf{N}}$
9:     $\forall i \in \mathsf{N}$ :
10:        $params.\mathcal{P}[\cdot].1^{\kappa} \leftarrow 1^{|params|}$
11:        $params.\mathcal{P}[\cdot].maxAOut \leftarrow s_{\mathsf{ACom}(\mathcal{A})}.maxAOut$
12:        $params.\mathcal{P}[\cdot].maxPOut \leftarrow s_{\mathsf{ACom}(\mathcal{A})}.maxPOut$
13:    return $(s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}}, \mathsf{N}, params.\mathcal{P}[\cdot])$
14: else
15:    $s_{\mathsf{ACom}(\mathcal{A})}, inpA \leftarrow inputs$
16:    if $inpA = \bot$ then
17:        $outA \leftarrow \mathcal{A}(s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}})$
18:        return $trunc(outA, s_{\mathsf{ACom}(\mathcal{A})}.maxAOut)$
19:    else
20:        $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}(s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}}, inpA)$
21:        $s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}} \leftarrow trunc(s_{\mathcal{A}}, s_{\mathsf{ACom}(\mathcal{A})}.maxStateSz)$
22:        $s_{\mathsf{ACom}(\mathcal{A})}.numEvents \leftarrow s_{\mathsf{ACom}(\mathcal{A})}.numEvents + 1$
23:        if $s_{\mathsf{ACom}(\mathcal{A})}.numEvents > s_{\mathsf{ACom}(\mathcal{A})}.maxStateSz$ then
24:            $out_{\mathcal{A}} \leftarrow \top$
25:        end if
26:        return $(s_{\mathsf{ACom}(\mathcal{A})}.s_{\mathcal{A}}, trunc((out_{\mathcal{A}}, \mathsf{F}), s_{\mathsf{ACom}(\mathcal{A})}.maxAOut))$
27:    end if
28: end if
```

The input to PCom is a (protocol) algorithm $\mathcal{P}$ and the output is a modified protocol algorithm $\mathsf{PCom}(\mathcal{P})$. Pseudocode for the $\mathsf{PCom}(\mathcal{P})$ algorithm is shown in Algorithm 11. The algorithm $\mathsf{PCom}(\mathcal{P})$ runs $\mathcal{P}$ internally on provided inputs and then 'processes' its outputs as follows:

– At initialization, $\mathsf{PCom}(\mathcal{P})$ saves the value of $|params.maxPOut|$ internally. After initialization, whenever $\mathcal{P}$ is run at an entity i, if the output of $\mathcal{P}$ is longer than the protocol's maximum output size, then $\mathsf{PCom}(\mathcal{P})$ truncates it to have size $|params.maxPOut|$. Similarly, if $\mathcal{P}$ outputs a new state for entity i that is longer than the entity's previous state, then $\mathsf{PCom}(\mathcal{P})$ truncates it to have the same size as the initial $|s_i|$. The $\mathsf{PCom}(\mathcal{P})$ algorithm outputs the truncated outputs and state to the execution process.

Algorithm 11 $\mathsf{PCom}(\mathcal{P})[opr](s_{\mathsf{PCom}(\mathcal{P})}, inp, clk)$

```
1: if $opr =$ 'Init' and $s_{\mathsf{PCom}(\mathcal{P})} = \perp$ then
2:     $params.\mathcal{P} \leftarrow inp$
3:     $s \leftarrow \mathcal{P}[\text{'Init'}](\perp, params.\mathcal{P}, \perp)$
4:     $s_{\mathsf{PCom}(\mathcal{P})}.s \leftarrow s$
5:     $s_{\mathsf{PCom}(\mathcal{P})}.maxStateSz \leftarrow |s|$
6:     $s_{\mathsf{PCom}(\mathcal{P})}.maxPOut \leftarrow params.\mathcal{P}.maxPOut$
7:     return $s$
8: else
9:     $(s, out, sec\text{-}out) \leftarrow \mathcal{P}[opr](s_{\mathsf{PCom}(\mathcal{P})}.s, inp, clk)$
10:    $s_{\mathsf{PCom}(\mathcal{P})}.s \leftarrow trunc(s, s_{\mathsf{PCom}(\mathcal{P})}.maxStateSz)$
11:    return $(s_{\mathsf{PCom}(\mathcal{P})}.s, trunc((out, sec\text{-}out), s_{\mathsf{PCom}(\mathcal{P})}.maxPOut))$
12: end if
```

Note: We believe that the above definitions of ACom and PCom could be altered. For example, it may be possible to allow the states to grow by a fixed amount (polynomial in $|params|$) in each event; however, we decided to use fixed state sizes for simplicity.

B.2 Model and Requirement Satisfaction Using ACom and PCom

We now use the ACom and PCom compilers to give alternate definitions of model-satisfying and requirement-satisfying, in which we consider executions with $\mathsf{ACom}(\mathcal{A})$ and $\mathsf{PCom}(\mathcal{P})$ instead of $\mathcal{A}$ and $\mathcal{P}$. In Sec. B.5, we show that for $\mathcal{A}, \mathcal{P} \in PPT$, such execution (with $\mathsf{ACom}(\mathcal{A})$ and $\mathsf{PCom}(\mathcal{P})$) with some restrictions on the $\mathcal{X}$-operations suffices to ensure polynomial runtime of both $\mathcal{A}$ and $\mathcal{P}$ during the execution.

Definition 6 (Adversary $\mathcal{A}$ polybound-satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$). *Let $\mathcal{A}, \mathcal{X}$ be algorithms and let $\mathcal{M}$ be a set of specifications, i.e., $\mathcal{M} = \{(\pi_1, \beta_1), \ldots\}$. We say that adversary $\mathcal{A}$ polybound-satisfies model $\mathcal{M}$ using execution operations $\mathcal{X}$, denoted $\mathcal{A} \models^{\mathcal{X}}_{polybound} \mathcal{M}$, if for every protocol $\mathcal{P}$, $params \in \{0,1\}^*$, and specification $(\pi, \beta) \in \mathcal{M}$, the advantage of $\mathsf{ACom}(\mathcal{A})$ against $\mathsf{PCom}(\mathcal{P})$ for π using $\mathcal{X}$ is at most negligibly greater than $\beta(params)$, i.e.:*

$$\mathcal{A} \models^{\mathcal{X}}_{polybound} \mathcal{M} \overset{def}{=} \left[\begin{array}{c} (\forall\ \mathcal{P} \in PPT, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}) : \\ \epsilon^{\pi}_{\mathsf{ACom}(\mathcal{A}), \mathsf{PCom}(\mathcal{P}), \mathcal{X}}(params) \leq \beta(params) + Negl(|params|) \end{array} \right] \tag{12}$$

Definition 7 (Protocol $\mathcal{P}$ polybound-satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$). *Let $\mathcal{P}, \mathcal{X}$ be algorithms, and let $\mathcal{R}$ be a set of specifications, i.e., $\mathcal{R} = \{(\pi_1, \beta_1), \ldots\}$. We say that protocol $\mathcal{P}$ polybound-satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$ using execution operations $\mathcal{X}$, denoted*

$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathit{polybound}} \mathcal{R}$*, if for every adversary* $\mathcal{A}$ *that polybound-satisfies* $\mathcal{M}$ *using execution operations* $\mathcal{X}$*, every* $\mathit{params} \in \{0,1\}^*$*, and every specification* $(\pi, \beta) \in \mathcal{R}$*, the advantage of* $\mathsf{ACom}(\mathcal{A})$ *against* $\mathsf{PCom}(\mathcal{P})$ *for* π *using* $\mathcal{X}$ *is at most negligibly greater than* $\beta(\mathit{params})$*, i.e.:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathit{polybound}} \mathcal{R} \overset{def}{=} \begin{bmatrix} (\forall\ \mathcal{A} \in PPT\ s.t.\ \mathcal{A} \models^{\mathcal{X}}_{\mathit{polybound}} \mathcal{M},\ \mathit{params} \in \{0,1\}^*,\ (\pi, \beta) \in \mathcal{R}) : \\ \epsilon^{\pi}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathcal{P}),\mathcal{X}}(\mathit{params}) \leq \beta(\mathit{params}) + Negl(|\mathit{params}|) \end{bmatrix} \tag{13}$$

B.3 Modeling Parameter Assumptions

Before continuing to *bounded* $\mathcal{X}$*-operations* and the Polynomial Runtime Theorem, we briefly discuss the use of the maximum input and output bounds, which are given by the adversary to the protocol in $\mathit{params}.\mathcal{P}[\cdot].\mathit{maxAOut}$ and $\mathit{params}.\mathcal{P}[\cdot].\mathit{maxPOut}$. Real-world protocols typically have some limits (polynomial in the security parameter) on the maximum inputs and output size. Thus, we use $\mathit{params}.\mathcal{P}[\cdot].\mathit{maxAOut}$ and $\mathit{params}.\mathcal{P}[\cdot].\mathit{maxPOut}$ to specify these bounds.

Recall that we allow the adversary to choose the maximum input and output sizes of the protocol, and we use the ACom and PCom compilers to enforce these limits. (Note that the ACom compiler 'automatically' ensures that all entities get the same values for these bounds. It also 'automatically' gives the security parameter $|\mathit{params}|$ to all entities in unary in the parameter $\mathit{params}.\mathcal{P}[\cdot].1^{\kappa}$.) Notice that this implies that if $\mathcal{A} \in PPT$, then the sizes of the maximum input and output bounds are polynomial in $|\mathit{params}|$. However, not all values for these bounds might make sense for a particular protocol, so it may be useful to restrict the allowed values using models, just like how other assumptions are specified in MoSS. Similarly, the adversary's behavior after choosing particular values for these bounds can be restricted using models. Below, we show such an example model, which is used in the analysis of the AuthBroadcast protocol (Sec. C).

In the analysis of the AuthBroadcast protocol, we want to assume that:

- The key length is at least as large as the tag length.
- The maximum input and output bounds are at least as large as the security parameter, so that keys can be sent and received by the protocol.
- The messages given to the protocol to be sent are not too long to be output properly with tags.
- The outgoing messages (with tags) output by the protocol are not too long to be received by the protocol.

For these assumptions, we can use the following model.

Algorithm 12 $\pi(T, params)$ Predicate

```
1: return ⊤ if (
2:     $n \leftarrow |params.1^n|$
3:     $\forall\ i \in T.\mathsf{N}$: $(T.params.\mathcal{P}[i].n = n))$
4:     and $\exists\ maxAOut, maxPOut$  s.t.  $\forall\ i \in T.\mathsf{N}:\ (T.params.\mathcal{P}[i].maxAOut = maxAOut$
                                              and $T.params.\mathcal{P}[i].maxPOut = maxPOut)$
                                          and $\forall \hat{e} \in \{1, \ldots, T.e\}$:
5:         if $T.opr[\hat{e}] =$ ‘Broadcast’ then:
               $|T.opr[\hat{e}]| + |T.inp[\hat{e}]| + |T.clk[\hat{e}]| + n \leq maxAOut$
               and $|T.opr[\hat{e}]| + |T.inp[\hat{e}]| + |T.clk[\hat{e}]| \leq maxPOut$
6:         else if $T.opr[\hat{e}] =$ ‘Key-Gen’ then:
               and $|params| \leq maxPOut$
7:         else if $T.opr[\hat{e}] =$ ‘Sec-in’ then:
               $|T.opr[\hat{e}]| + |T.inp[\hat{e}]| + |T.clk[\hat{e}]| + |params| \leq maxAOut$
   )
```

In the model predicate shown in Algorithm 12, line 3 ensures that the tag length is set to $|params.1^n|$ for all entities - that is, we assume that some parameter 1^n is given as part of *params* to the execution process (and the adversary), and the adversary sets the tag length for the protocol to be the size of this parameter. Since the parameter is part of *params*, then the tag length is smaller than the security parameter $params.\mathcal{P}[\cdot].1^\kappa$ given to the protocol, which is used as the key length by the AuthBroadcast protocol.

Line 5 ensures that messages given by $\mathcal{A}$ to $\mathcal{P}$ to be broadcast (to be sent out) are short enough to be able to be given to $\mathcal{P}$ as *input* and also short enough to be able to be sent *out with tags* and delivered back to $\mathcal{P}$ as input with tags.

Line 6 ensures that the output bound of the protocol is large enough to allow a key of length $|params|$ to be sent out (through a secure channel).

Line 7 ensures that in ‘Sec-in’ operations, the inputs given by $\mathcal{A}$ to $\mathcal{P}$ are short enough to be able to be given to $\mathcal{P}$ as input *together with a key of length* $|params|$ (which would come from a secure channel).

Thus, using a model predicate, we can assume that the lengths of messages and other inputs which the adversary gives to the protocol are ‘reasonable’ with respect to the protocol’s maximum input and output sizes and the particular protocol. In effect, if we also assume a guaranteed delivery model, then we can assume that any ‘corrupted’ protocol outputs are really ‘corrupted’, not simply due to truncation by the compilers.

B.4 Bounded $\mathcal{X}$-operations

We will soon show that we can ensure polynomial runtime for executions in which we run ACom($\mathcal{A}$) and PCom($\mathcal{P}$) instead of $\mathcal{A}$ and $\mathcal{P}$ directly. First, however, we need to make sure that any $\mathcal{X}$-operations which are used in the execution also have accordingly bounded outputs and inputs to the protocol. For this purpose, we define a *bounded $\mathcal{X}$-operation*.

Definition 8 (Bounded $\mathcal{X}$-operation). *Let 'X' be an $\mathcal{X}$-operation. We say that 'X' is a* bounded $\mathcal{X}$-operation *if:*

- *'X' does not run the adversary.*
- *'X' runs the protocol at most once.*
- *If 'X' runs the protocol for entity i, then the input (not including state) which it gives to the protocol has size at most $params.maxAOut$ and the input state which it gives to the protocol has size $|s_i|$.*
- *When 'X' is run for entity i, the output size (not including state) of 'X' is at most $params.maxPOut$ and the output state for the entity has size $|s_i|$.*

In short, a bounded $\mathcal{X}$-operation limits the size of inputs to the protocol and outputs of the operation in the way that $\mathsf{ACom}(\mathcal{A})$ and $\mathsf{PCom}(\mathcal{P})$ do.

We can easily define variants of the previously defined $\mathcal{X}$-operations ('Flip', 'Challenge', 'Guess', 'Sec-in', 'Get-state', 'Set-output', 'Set-state') which are bounded $\mathcal{X}$-operations. They would work the same, except they would truncate the inputs to the protocol and/or outputs of the $\mathcal{X}$-operations if needed; e.g., the modified 'Sec-in' would give to the protocol a truncation of the output from a previous event and part of the adversary's input from the current event; the modified 'Set-state' would output a truncation of the state provided by the adversary as the new state of the entity; etc.

B.5 The Polynomial Runtime Theorem

We now present the Polynomial Runtime Theorem, which motivates the use of the ACom and PCom compilers and bounded $\mathcal{X}$-operations.

Theorem 1 (Polynomial Runtime Theorem). *Let $\mathcal{A}, \mathcal{P}, \mathcal{X} \in PPT$ such that all operations in $\mathcal{X}$ are bounded $\mathcal{X}$-operations. Then in a MoSS execution $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathcal{P})}(params)$, the runtime of $\mathcal{A}$ and the runtime of $\mathcal{P}$ are bounded by polynomials in $|params|$.*

Proof. $\mathcal{A}$ is a PPT algorithm, so its runtime during initialization (that is, when $\mathcal{A}[\text{'Init'}](params)$ is run) must be bounded by some polynomial $p(|params|)$ in the security parameter $|params|$. Notice that since $params.\mathcal{P}[\cdot].maxAOut$ and $params.\mathcal{P}[\cdot].maxPOut$ are output by $\mathcal{A}$ when it is initialized, then their lengths are also bounded by $p(|params|)$. The size of the initial state output by the adversary, $|s_{\mathcal{A}}|$, is bounded by $p(|params|)$ as well, since it is output when $\mathcal{A}$ is initialized. Due to $\mathsf{ACom}(\mathcal{A})$, this implies that the number of events is bounded by $p(|params|)$.

$\mathsf{ACom}(\mathcal{A})$ and bounded $\mathcal{X}$-operations ensure that the size of input, excluding state, to the protocol in any event is bounded by $|params.maxAOut|$ and hence by $p(|params|)$. The protocol also receives the entity's state. Recall that $\mathsf{PCom}(\mathcal{P})$ and bounded $\mathcal{X}$-operations ensure that the size of the state of each entity stays the same as it was initially; the state of each entity i is initially output by $\mathcal{P}[\text{'Init'}](\bot, params.\mathcal{P}[i], \bot)$; and $|params.\mathcal{P}[i]|$ is bounded by some polynomial in $|params|$, since it is output by the adversary when it is initialized and then only some values are overwritten by $\mathsf{ACom}(\mathcal{A})$ to give the security parameter to the protocol and to make sure that the input and output bounds are the same for all entities. Then, since $\mathcal{P} \in PPT$, $|s_i|$ is bounded by some polynomial in $|params|$. Thus, the size of input to $\mathcal{P}$ in any event is bounded by some polynomial in $|params|$. This implies that the runtime of $\mathcal{P}$ in any event is bounded by a polynomial, and since the number of events is bounded by $p(|params|)$, then the runtime of $\mathcal{P}$ over all events is bounded by a polynomial in $|params|$.

The input to the adversary is at most the adversary's state and the output of $\mathsf{PCom}(\mathcal{P})$, which are both bounded by a polynomial in $|params|$ due to $\mathsf{ACom}(\mathcal{A})$, $\mathsf{PCom}(\mathcal{P})$, and bounded $\mathcal{X}$-operations. Thus, the input size to $\mathcal{A}$ is bounded by a polynomial and therefore the runtime of $\mathcal{A}$ whenever it is run is bounded by a polynomial. Since the number of events is bounded by $p(|params|)$, then the runtime of $\mathcal{A}$ in the whole execution is bounded by a polynomial in $|params|$.

□

C 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.

C.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 §C.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 20): initializes the local state of the entity.
- 'Key-Gen' (Algorithm 21): generates the shared authentication key and sends it to the rest of the entities.
- 'Sec-in' (Algorithm 22): receives a shared-key from another entity.
- 'Broadcast' (Algorithm 23): broadcasts an authenticated message.
- 'Receive' (Algorithm 24): receives incoming broadcast message, verifies its authenticity and outputs it (to the application).

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

C.2 AuthBroadcast Models

We define three increasingly stronger models in Sec C.2.1, C.2.2, and C.2.3, respectively, each a union of multiple sub-models. The first model includes secure key sharing assumptions, the second includes secure key sharing and bounded clock drift assumptions, and the third includes secure key sharing, bounded clock drift, and bounded-delay communication assumptions. Specifically, the three models $\mathcal{M}_{\text{SecKeyInit}}$, $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$, and $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ are defined as follows.

$$\begin{aligned}
\mathcal{M}_{\text{SecKeyInit}} &\equiv \mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]} \cup \mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{`Sec-in'}]} \cup \mathcal{M}^{\kappa\geq n}_{params} \\
\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}} &\equiv \mathcal{M}_{\text{SecKeyInit}} \cup \mathcal{M}^{\text{Drift}}_{\Delta_{clk}} \\
\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}} &\equiv \mathcal{M}_{\text{SecKeyInit}} \cup \mathcal{M}^{\text{Drift}}_{\Delta_{clk}} \cup \mathcal{M}^{\text{Broadcast}}_{\Delta_{com}} \cup \mathcal{M}^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}
\end{aligned}$$

for $f(\Delta_{com}, \Delta_{clk}) = \Delta_{com} + 2\Delta_{clk}$. Later, we analyze the $\mathsf{AuthBroadcast}$ protocol under each of these three models.

C.2.1 Secure Key Sharing Model

The first model, $\mathcal{M}_{\text{SecKeyInit}}$, is a model that assumes secure key sharing and valid n and 1^κ parameters given to the protocol. It is defined as:

$$\mathcal{M}_{\text{SecKeyInit}} \equiv \mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]} \cup \mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{`Sec-in'}]} \cup \mathcal{M}^{\kappa\geq n}_{params}$$

We next define the three sub-models, $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$, $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{`Sec-in'}]}$, and $\mathcal{M}^{\kappa\geq n}_{params}$.

The $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ model is defined as $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]} = (\pi^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}, 0)$, where the $\pi^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ predicate (Algorithm 13) 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 $\pi^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ 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 $\pi^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ 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 use this model for its 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 13 $\pi^{\text{KeyShare}}_{\mathcal{X}[\text{‘Sec-in’}]}(T, \mathit{params})$ 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 needed for a ‘Broadcast’ or ‘Receive’ event
           then $\exists\ \hat{e}'' \in \{\hat{e}+1, \ldots, \hat{e}'-1\}$     ▷ Then prior to the event
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 $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{‘Sec-in’}]}$ model is defined as $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{‘Sec-in’}]} = (\pi^{\text{Exclude}}_{\mathcal{P}[\text{‘Sec-in’}]}, 0)$, where the $\pi^{\text{Exclude}}_{\mathcal{P}[\text{‘Sec-in’}]}$ predicate makes sure that the adversary does not cause an entity to receive ‘fake’ securely shared values (using the ‘Sec-in’ operation). It 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 14 $\pi^{\text{Exclude}}_{\mathcal{P}[\text{‘Sec-in’}]}(T, \mathit{params})$ 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 $\mathcal{M}^{\kappa \geq n}_{params}$ model is defined as $\mathcal{M}^{\kappa \geq n}_{params} = (\pi^{\kappa \geq n}_{params}, 0)$, where the $\pi^{\kappa \geq n}_{params}$ predicate is shown in Algorithm 15. For the AuthBroadcast protocol, the $\pi^{\kappa \geq n}_{params}$ predicate ensures that the key used by the protocol for authentication (using a MAC) is at least as long as the tags. Specifically, the predicate checks that for all $i \in \mathsf{N}$, the adversary sets $params.\mathcal{P}[i].n$ to $|params.1^n|$ and sets $params.\mathcal{P}[i].1^\kappa$ to $1^{|params|}$. The values $params.\mathcal{P}[i].n$ and $params.\mathcal{P}[i].1^\kappa$ are used to initialize the AuthBroadcast protocol at entity i. Consequently, the MAC can receive sufficient randomness to generate secure n-bit tags for authentication of messages, where $n = |params.1^n|$.

Algorithm 15 $\pi^{\kappa \geq n}_{params}(T,\, params)$ Predicate

```
1: return ⊤ if (
2:     $\forall\, i \in \mathsf{N}:\ (T.params.\mathcal{P}[i].n = |params.1^n|)$ and $(T.params.\mathcal{P}[i].1^\kappa = 1^{|params|})$
   )
```

Thus, the $\mathcal{M}_{\text{SecKeyInit}}$ model assumes that a key is securely shared once among all entities and that valid n and 1^κ parameters are given to the protocol.

C.2.2 Secure Key Sharing and Bounded Clock Drift Model

The second model, $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$, assumes bounded clock drift in addition to the secure key sharing and valid n and 1^κ parameters given to the protocol assumed by $\mathcal{M}_{\text{SecKeyInit}}$. It is defined as:

$$\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}} \equiv \mathcal{M}_{\text{SecKeyInit}} \cup \mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$$

where $\mathcal{M}_{\text{SecKeyInit}}$ is defined in Sec. C.2.1 and $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ is defined in Sec. 3.3. Thus, the $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ model assumes that:

1. A key is securely shared once among all entities, and valid n and 1^κ parameters are given to the protocol ($\mathcal{M}_{\text{SecKeyInit}}$).
2. Real time is monotonically increasing, and local time at all entities is always within Δ_{clk} drift from the real time ($\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$).

C.2.3 Secure Key Sharing, Bounded Clock Drift, and Bounded-Delay Communication Model

The third model, $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$, assumes bounded-delay communication and a sufficiently large freshness interval parameter given to the protocol, in addition to the secure key sharing, valid n and 1^κ parameters given to the protocol, and bounded clock drift assumed by $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$. It is defined as:

$$\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}} \equiv \mathcal{M}_{\text{SecKeyInit}} \cup \mathcal{M}^{\text{Drift}}_{\Delta_{clk}} \cup \mathcal{M}^{\text{Broadcast}}_{\Delta_{com}} \cup \mathcal{M}^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$$

for $f(\Delta_{com}, \Delta_{clk}) = \Delta_{com} + 2\Delta_{clk}$. The $\mathcal{M}_{\text{SecKeyInit}}$ model is defined in Sec. C.2.1 and $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ is defined in Sec. 3.3. We next define the $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}}$ and $\mathcal{M}^{f \leq \Delta}_{\Delta_{com},\Delta_{clk}}$ sub-models.

The $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}}$ model $\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}} = (\pi^{\text{Broadcast}}_{\Delta_{com}}, 0)$, where the $\pi^{\text{Broadcast}}_{\Delta_{com}}$ predicate, shown in Algorithm 16, verifies that every 'Broadcast' packet 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). Intuitively, the model assumes that the communication channel delivers 'Broadcast' packets to all other entities.

Algorithm 16 $\pi^{\text{Broadcast}}_{\Delta_{com}}$ (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 $\forall i \in T.\mathsf{N}$ 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 $\mathcal{M}^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}$ model is defined as $\mathcal{M}^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}} = (\pi^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}, 0)$. The $\pi^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}$ model predicate, shown in Algorithm 17, checks that for every entity i, the parameter $params.\mathcal{P}[i].\Delta$ (which is used as the freshness interval in the AuthBroadcast protocol) is at least as large as some given function f of Δ_{com} and Δ_{clk}, which are the assumed maximal communication delay and assumed maximal clock drift, respectively. 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 $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}[i].\Delta$ for each entity i. Intuitively, this is necessary for the AuthBroadcast protocol to provide guaranteed delivery, because otherwise, although a message packet may arrive, the receiver's Δ parameter (freshness interval) may be too small to consider the packet fresh based on its timestamp. Then, the packet would be considered old by the protocol and would not be received successfully.

Algorithm 17 $\pi^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}$ (T, $params$) Predicate

```
1: return ($\forall\ i \in T.\mathsf{N}:\ f(\Delta_{com}, \Delta_{clk}) \leq params.\mathcal{P}[i].\Delta$)
```

Therefore, the $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ model assumes that:

1. A key is securely shared once among all entities, and valid n and 1^κ parameters are given to the protocol ($\mathcal{M}_{\text{SecKeyInit}}$).
2. Real time is monotonically increasing, and local time at all entities is always within Δ_{clk} drift from the real time ($\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$).
3. There is reliable, bounded-delay broadcast communication ($\mathcal{M}^{\text{Broadcast}}_{\Delta_{com}}$).
4. $\Delta_{com}+2\Delta_{clk} \leq params.\mathcal{P}[i].\Delta$ for every entity i, which is needed for the protocol to ensure receipt/delivery of valid ‘Broadcast’ packets ($\mathcal{M}^{f\leq\Delta}_{\Delta_{com},\Delta_{clk}}$).

C.3 AuthBroadcast Security Requirements

We define three requirements, $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{\infty}}$, $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$, and $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta}}$, in Sec. C.3.1, C.3.2, and C.3.3, respectively. The first ensures authenticity of received broadcast messages, the second ensures authenticity and freshness, and the third ensures correct bounded-delay delivery/receipt of all broadcast messages (which implies authenticity and freshness as well). Later, we show that the AuthBroadcast protocol polybound-satisfies (as defined in Sec. B.2) these three requirements under the models defined in Sec. C.2.1, C.2.2, and C.2.3, respectively.

C.3.1 Authentic Broadcast Requirement

The first requirement, $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{\infty}}$, ensures authenticity of received broadcast messages and is defined as:

$$\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{\infty}} \equiv \{(\pi^{\text{Broadcast}}_{\text{Auth}_{\infty}}, \beta^{\text{Broadcast}}_{\text{Auth}_{\infty}})\}$$

where the predicate $\pi^{\text{Broadcast}}_{\text{Auth}_{\infty}}$ is the predicate $\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ defined in Sec. C.3.2, for $f(\Delta) = \infty$, and the base function is defined as $\beta^{\text{Broadcast}}_{\text{Auth}_{\infty}}(params) = 2^{-|params.1^n|}$. The base function is $2^{-|params.1^n|}$ because the AuthBroadcast protocol is implemented using some message authentication scheme MAC and $|params.1^n|$ is assumed to be given as the tag length parameter to AuthBroadcast (this is assumed by the $\mathcal{M}^{\kappa \geq n}_{params}$ model, which is part of all three models later used for analysis of AuthBroadcast). Thus, we allow the adversary to have $2^{-|params.1^n|}$ probability to forge a tag, which would make $\pi^{\text{Broadcast}}_{\text{Auth}_{\infty}}$ evaluate to $\perp$. Of course one could change to negligible probability by using a model which ensures that the adversary sets $params.\mathcal{P}[i].n$ and $params.\mathcal{P}[i].1^{\kappa}$ to be the same.

Thus, the $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{\infty}}$ requirement ensures authentication (but not freshness, since ∞ is used for the function of the freshness interval in the predicate).

C.3.2 Authentic, Fresh Broadcast Requirement

The second requirement, $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$, ensures authenticity and freshness of received broadcast messages and is defined as:

$$\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}} \equiv \{(\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}, \beta^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}})\}$$

where the base function is defined as $\beta^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}(params) = 2^{-|params.1^n|}$, for the same reasons as the $\beta^{\text{Broadcast}}_{\text{Auth}_{\infty}}$ base function discussed in Sec. C.3.1. We define the predicate $\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ next.

The $\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ predicate, shown in Algorithm 18, verifies that in a given protocol, messages that are received were actually broadcast (authentication) within the last $f(\Delta)$ real time (freshness), where $f(\Delta)$ is a function of the receiving entity's Δ parameter (which may be used as a freshness interval by a protocol; note that this value may be different for different entities). In the analysis of the AuthBroadcast protocol in Sec. C.6, in Claim 2 of Theorem 2, we use $f(\Delta) = \Delta + 2\Delta_{clk}$, since we assume a maximal clock drift of Δ_{clk}.

Algorithm 18 $\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}(T, params)$ 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 $T.\tau[\hat{e}_R] - T.\tau[\hat{e}_B] \leq f(T.params.\mathcal{P}[T.ent[\hat{e}_R]].\Delta)$   ▷ Within the last real time interval determined by f and the recipient's freshness interval
7:             and $T.inp[\hat{e}_B].m = m$            ▷ Where the input message was m
   )
```

Thus, the $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ requirement ensures authentication and freshness - i.e., all broadcast messages that are received and output as valid by the protocol were actually broadcast within the last $f(\Delta)$ real time, where Δ is the receiving entity's Δ parameter.

C.3.3 Bounded-delay Broadcast Requirement

The third requirement, $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_\Delta}$, ensures correct bounded-delay delivery/receipt of all broadcast messages, which implies authenticity and freshness as well. It is defined as:

$$\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_\Delta} \equiv \{(\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}, 0)\}$$

where the $\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}$ predicate, shown in Algorithm 19, requires that all broadcast messages are correctly delivered/received at every other entity (except the sender of the broadcast) within Δ real time (unless the execution terminates before that time, of course).

Algorithm 19 $\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}(T, params)$ 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:       $T.\tau[T.e] \geq T.\tau[\hat{e}_B] + \Delta$):        ▷ Which was at least $\Delta$ real time before the end of the execution
5:         $\forall i \in T.\mathsf{N}$ 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 $T.\tau[\hat{e}_B] + \Delta \geq T.\tau[\hat{e}_R]$        ▷ Within $\Delta$ 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
   )
```

C.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 20-24. 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 20 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Init'}](s, inp, clk)$

```
1: $params \leftarrow inp$
2: if $s = \bot$ then        ▷ This is the first call to 'Init'
3:     $1^\kappa \leftarrow params.1^\kappa$        ▷ Initialize key length
4:     $n \leftarrow params.n$        ▷ Initialize tag length
5:     $\Delta \leftarrow params.\Delta$        ▷ 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 21 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Key-Gen'}](s, inp, clk)$

```
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 22 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Sec-in'}](s, inp, clk)$

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

Algorithm 23 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Broadcast'}](s, inp, clk)$

```
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:     $out \leftarrow (\text{'Broadcast'}, m, timeSent, tag)$          ▷ Return 'Broadcast', the message, local time, and tag
6: end if
7: return $(s, out, \bot)$
```

Algorithm 24 $\mathsf{AuthBroadcast}^{\mathsf{MAC}}[\text{'Receive'}](s, inp, clk)$

```
1: $(m, timeSent, tag) \leftarrow inp$ ; $out \leftarrow \bot$
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:         $out \leftarrow (\text{'Receive'}, m)$          ▷ If all Ok, output m
6: end if
7: return $(s, out, \bot)$
```

C.5 Message Authentication Scheme

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

Definition 9. *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 10 (Asymptotically Unforgeable). *We call a message authentication scheme* MACasymptotically unforgeable (asymptotically UF) *if for every PPT adversary* $\mathcal{A}$ *and every* $\kappa, n \in \mathbb{N}$ *such that* $\kappa \geq n$ *(where* κ *is the the key length and* n *is the tag length), the probability that* $\mathcal{A}$ *'wins' the game* $\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(\kappa, n)$ *is at most negligibly greater (in* κ*) than* 2^{-n}*, i.e.:*

$$\begin{gathered}(\forall\ \mathcal{A} \in PPT \text{ and } \kappa, n \in \mathbb{N} \mid \kappa \geq n): \\ \Pr[\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(\kappa, n) = \top] \leq 2^{-n} + Negl(\kappa)\end{gathered} \tag{14}$$

where $\mathbf{Exp}^{UF}_{\mathcal{A},\mathsf{MAC}}(\kappa, n)$ *is defined in Algorithm 25.*

Algorithm 25 $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A},\mathsf{MAC}}(\kappa, n)$

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

C.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 AuthBroadcast in several steps, where in each step, we prove that AuthBroadcast satisfies a requirement - assuming increasingly stronger models:

1. We first show that AuthBroadcast ensures *authentication* of received messages assuming that a key is shared securely once among all entities and valid n and 1^{κ} parameters are given to the protocol. Namely, we show that AuthBroadcast polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{\infty}}$ under $\mathcal{M}_{\text{SecKeyInit}}$ using $\mathcal{X}$-operations {'Sec-in'}.
2. We then show that AuthBroadcast ensures *authentication and freshness* of received messages under a stronger model that also assumes a weak-level of clock synchronization (bounded clock drift). Namely, we show that AuthBroadcast polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$-operations {'Sec-in'} for $f(\Delta) = \Delta + 2\Delta_{clk}$, where Δ_{clk} is the assumed maximal clock drift.
3. Finally, we show that AuthBroadcast ensures *correct bounded-delay delivery/receipt* of broadcast messages (which implies authenticity and freshness as well) under an even stronger model which also assumes a bounded delay of communication and a sufficiently large freshness interval given to the protocol. Specifically, we show that AuthBroadcast polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ using $\mathcal{X}$-operations {'Sec-in'}, where Δ_{clk} is the assumed maximal clock drift and Δ_{com} is the assumed maximal communication delay.

Where the notion of 'polytime-satisfying' a requirement is defined in Definition 7 in Sec. B.2.

The reader may make the following two observations. First, the models used for the analysis of AuthBroadcast are composed of multiple sub-models which are *generic* to varying degrees. For example, the $\mathcal{M}_{\text{SecKeyInit}}$ model is defined as $\mathcal{M}_{\text{SecKeyInit}} \equiv \mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{'Sec-in'}]} \cup \mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{'Sec-in'}]} \cup \mathcal{M}^{\kappa \geq n}_{params}$, where the models $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ and $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{`Sec-in'}]}$ enforce the secure key sharing assumption and the model $\mathcal{M}^{\kappa \geq n}_{params}$ enforces the assumption that valid n and 1^κ parameters are given to the protocol. These are generic, since $\mathcal{M}^{\text{KeyShare}}_{\mathcal{X}[\text{`Sec-in'}]}$ and $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{`Sec-in'}]}$ can be reused for analysis of other shared-key protocols, and $\mathcal{M}^{\kappa \geq n}_{params}$ can be reused for analysis of other protocols which assume that some parameter is bounded by the length of some parameter. Second, by Lemma 3 (Sec. 5), it follows that $\mathsf{AuthBroadcast}$ also polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$, polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$, and polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$, since $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ is stronger than $\mathcal{M}_{\text{SecKeyInit}}$ and $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ is stronger than $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$. These two observations show some of the modular qualities of the MoSS framework.

Theorem 2. *Let* MAC *be an asymptotically unforgeable message authentication scheme, as defined in Appendix C.5. Let* $\mathcal{X}$ *be* $\{$*'Sec-in'*$\}$*, where 'Sec-in' is a bounded* $\mathcal{X}$*-operation (see Sec. 2.3 for discussion of 'Sec-in' and Sec. B.4 for discussion of bounded* $\mathcal{X}$*-operations). Then:*

Claim 1. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ *polybound-satisfies* $\mathcal{R}^{Broadcast}_{Auth_\infty}$ *under* $\mathcal{M}_{SecKeyInit}$ *using* $\mathcal{X}$*, i.e.:*

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}_{SecKeyInit},\, \mathcal{X}}_{polybound} \mathcal{R}^{Broadcast}_{Auth_\infty}$$

Where $\mathcal{M}_{SecKeyInit}$ *is defined in Sec. C.2.1 and* $\mathcal{R}^{Broadcast}_{Auth_\infty}$ *is defined in Sec. C.3.1.*

Claim 2. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ *polybound-satisfies* $\mathcal{R}^{Broadcast}_{Auth_{f(\Delta)}}$ *under* $\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}}}$ *using* $\mathcal{X}$ *for* $f(\Delta) = \Delta + 2\Delta_{clk}$*, i.e.:*

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}}},\, \mathcal{X}}_{polybound} \mathcal{R}^{Broadcast}_{Auth_{f(\Delta)}}$$

for $f(\Delta) = \Delta + 2\Delta_{clk}$*, where* $\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}}}$ *is defined in Sec. C.2.2 and* $\mathcal{R}^{Broadcast}_{Auth_{f(\Delta)}}$ *is defined in Sec. C.3.2.*

Claim 3. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ *polybound-satisfies* $\mathcal{R}^{Broadcast}_{Receive_{\Delta_{com}}}$ *under* $\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}},Delay_{\Delta_{com}}}$ *using* $\mathcal{X}$*-operations* $\{$*'Sec-in'*$\}$*, i.e.:*

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}},Delay_{\Delta_{com}}},\, \mathcal{X}}_{polybound} \mathcal{R}^{Broadcast}_{Receive_{\Delta_{com}}}$$

Where $\mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}},Delay_{\Delta_{com}}}$ *is defined in Sec. C.2.3 and* $\mathcal{R}^{Broadcast}_{Receive_{\Delta_{com}}}$ *is defined in Sec. C.3.3.*

C.6.1 Proof of Claim 1

In this section, we prove Claim 1 of Theorem 2 - that $\mathsf{AuthBroadcast}$ ensures authentication of received messages assuming that a key is shared securely once among all entities and valid n and 1^κ parameters are given to the protocol. The claim is restated below, for an asymptotically unforgeable message authentication scheme MAC (as defined in Appendix C.5) and $\mathcal{X} = \{\text{'Sec-in'}\}$, where 'Sec-in' is a bounded $\mathcal{X}$-operation (see Sec. 2.3 for discussion of 'Sec-in' and Sec. B.4 for discussion of bounded $\mathcal{X}$-operations).

Claim 1. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}_{\text{SecKeyInit}}$ using $\mathcal{X}$, i.e.:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}_{\text{SecKeyInit}},\,\mathcal{X}}_{\text{polybound}} \mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$$

Where $\mathcal{M}_{\text{SecKeyInit}}$ is defined in Sec. C.2.1 and $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ is defined in Sec. C.3.1.

We prove Claim 1 by contradiction - namely, by showing that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}_{\text{SecKeyInit}}$ using $\mathcal{X}$, then MAC is not asymptotically unforgeable.

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 $\overline{\mathsf{AuthBroadcast}}$. We also define an algorithm $\mathcal{A}''$, which is used by $\mathcal{A}'$.
2. Then we show that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$ under $\mathcal{M}_{\text{SecKeyInit}}$ using $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A}$ such that $\mathcal{A} \models^{\mathcal{X}}_{\text{polybound}} \mathcal{M}_{\text{SecKeyInit}}$ and $\mathcal{A}$ has probability non-negligibly (in the size of input parameters) greater than 2^{-n} to 'win' against the $\overline{\mathsf{AuthBroadcast}}$ protocol (w.r.t requirement $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_\infty}$), where n is a tag length parameter.
3. Lastly, we show that $\mathcal{A}''$ can 'win' the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ with probability greater than $2^{-n} + Negl(\kappa)$ for some $\kappa \geq n$, which implies that MAC is not an asymptotically unforgeable message authentication scheme.

Step 1: Protocol $\overline{\mathsf{AuthBroadcast}}$, Adversary $\mathcal{A}'$, and Subroutine $\mathcal{A}''$

$\mathcal{A}'$ takes $params \in \{0,1\}^*$ as input and works as follows:

1. $\mathcal{A}'$ modifies the code for $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ into $\overline{\mathsf{AuthBroadcast}}$. Specifically, line 4 in the 'Broadcast' function of $\mathsf{AuthBroadcast}$ (see Algorithm 23) is replaced by:

   $$tag \leftarrow \overline{\mathsf{Tag}}(m \mid\mid timeSent)$$

   $\mathcal{A}'$ also changes line 3 in the 'Receive' function of $\mathsf{AuthBroadcast}$ (see Algorithm 24) to:

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

$\mathcal{A}'$ also adds the following line after line 6 in the 'Init' function of $\mathsf{AuthBroadcast}$ (see Algorithm 20):

$$\overline{\mathsf{Init}}(1^\kappa, n)$$

$\overline{\mathsf{Tag}}$ and $\overline{\mathsf{Ver}}$ are initially placeholders. When $\overline{\mathsf{Init}}(1^\kappa, n)$ is invoked, it passes the values κ and n to $\mathcal{A}'$ and starts the game $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ (see Alg. 25) if it is not already started, where $\mathcal{A}''$ is an algorithm controlled by $\mathcal{A}'$. When the game gets to line 5 (in Alg. 25), $\mathcal{A}''$ gets access to oracles OTag and OVer. It then makes $\overline{\mathsf{Tag}}$ point to OTag and $\overline{\mathsf{Ver}}$ point to OVer. Note that $\mathsf{OTag}(\cdot)$ is an oracle that takes a message m as input and returns $\mathsf{MAC}_n(k, m)$, and $\mathsf{OVer}(\cdot, \cdot)$ is an oracle that takes inputs m and tag and returns $\top$ if $\mathsf{MAC}_n(k, m) = tag$ and $\perp$ otherwise, where k is a key chosen uniformly from $\{0, 1\}^\kappa$ and used by both oracles.

2. $\mathcal{A}'$ executes $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, where $\mathcal{A}$ is discussed below. During the execution, as long as there is at least one entity, $\overline{\mathsf{AuthBroadcast}}$['Init'] is invoked and the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ is started as described above.
3. $\mathcal{A}'$ searches T for 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. If $\mathcal{A}'$ finds such an event, it passes $T.inp[\hat{e}_{\mathrm{R}}].m \;||\; T.inp[\hat{e}_{\mathrm{R}}].timeSent, T.inp[\hat{e}_{\mathrm{R}}].tag$ to $\mathcal{A}''$, which outputs this to the (still ongoing) $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game; otherwise it passes a pair of randomly chosen strings $x \xleftarrow{R} \{0, 1\}^n, tag \xleftarrow{R} \{0, 1\}^n$, to $\mathcal{A}''$, which outputs x, tag to the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game.

Step 2: Adversary $\mathcal{A}$

From Definition 7, if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}$ under $\mathcal{M}_{\mathrm{SecKeyInit}}$ using $\mathcal{X} = \{$'Sec-in'$\}$ $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\mathrm{polybound}} \mathcal{M}_{\mathrm{SecKeyInit}}$ such that:

$$\epsilon^{\pi^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) > \beta^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}(params) + Negl(|params|) \tag{15}$$

where:

$$\epsilon^{\pi^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) \equiv \Pr\left[\begin{array}{c} \pi^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}(T, params) = \perp, \text{ where} \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}})}(params) \end{array}\right]$$

and $\beta^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}(params) = 2^{-|params.1^n|}$.

We show now that Equation 15 implies Equation 16, as stated in Lemma 6 below. Note that the difference between Eq. 15 and Eq. 16 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 6. *Let $\mathcal{X} = \{\text{'Sec-in'}\}$. Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{polybound} \mathcal{M}_{SecKeyInit}$ which satisfies Equation 15. Then holds:*

$$\epsilon^{\pi^{Broadcast}_{Auth_\infty}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}}),\mathcal{X}}(params) > \beta^{Broadcast}_{Auth_\infty}(params) + Negl(|params|) \tag{16}$$

Where:

$$\epsilon^{\pi^{Broadcast}_{Auth_\infty}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}}),\mathcal{X}}(params) \equiv \Pr\left[\begin{array}{c} \pi^{Broadcast}_{Auth_\infty}(T, params) = \bot,\ where \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params) \end{array}\right]$$

and $\beta^{Broadcast}_{Auth_\infty}(params) = 2^{-|params.1^n|}$.

Proof of Lemma 6.

From Equation 15, $\mathcal{A}$ is a PPT adversary that is able to cause the $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ protocol to correctly receive a message without previously broadcasting that message (with probability $> 2^{-|params.1^n|} + Negl(|params|)$). Recall that, according to Alg. 23, the 'Broadcast' function of the protocol outputs the message m, local time $timeSent$, and tag $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent)$, where $s.k$ is unknown to $\mathcal{A}$ due to secure key sharing (except for only negligible probability, as ensured by $\mathcal{M}_{\text{SecKeyInit}}$) and $s.n = |params.1^n|$ due to $\mathcal{M}^{\kappa \geq n}_{params} \in \mathcal{M}_{\text{SecKeyInit}}$ (except for negligible probability). Also recall that, according to Alg. 24, the 'Receive' function of the protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent) = tag$. This means that, with probability $> 2^{-|params.1^n|} + Negl(|params|)$, $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag such that $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent) = tag$.

Executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$ only differ from executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}})}(params)$ by the added call to $\overline{\mathsf{Init}}$ in 'Init', which does not alter the state or the outputs of 'Init', and the changes to 'Broadcast' and 'Receive', which cause messages to be authenticated using the OTag and OVer oracles in executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, instead of using $\mathsf{MAC}_{s.n}(s.k, \cdot)$, where s is the state of some entity and $s.k$ is a key chosen uniformly from $\{0,1\}^{|s.1^\kappa|}$, shared securely among the entities, and unknown to $\mathcal{A}$. But $\mathsf{OTag}(\cdot)$ uses $\mathsf{MAC}_n(k, \cdot)$ to compute a tag over a message, $\mathsf{OVer}(\cdot, \cdot)$ uses $\mathsf{MAC}_n(k, \cdot)$ to verify that a tag is correct, and both oracles use the same key chosen uniformly from $\{0,1\}^\kappa$, where n is the tag length and κ is the key length used by the entities. Thus, the executions should be equivalent to $\mathcal{A}$, which implies Equation 16.

□

Step 3: Completing the proof of Claim 1

We can now complete the proof of Claim 1. That is, we show that adversary $\mathcal{A}''$ wins the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game with probability greater than $2^{-n} + Negl(\kappa)$ for $\kappa \geq n$.

Proof. Suppose that $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\mathrm{Broadcast}}_{\mathrm{Auth}_\infty}$ under $\mathcal{M}_{\mathrm{SecKeyInit}}$ using $\mathcal{X}$. Then Equation 15 holds (from Definition 7).

By Lemma 6, Equation 16 holds. This means that when $\mathcal{A}'$ runs $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, $\mathcal{A}'$ has probability $> 2^{-|params.1^n|} + Negl(|params|)$ to find that T contains 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. Whenever this is the case, $\mathcal{A}'$ gives $T.inp[\hat{e}_{\mathrm{R}}].m \,||\, T.inp[\hat{e}_{\mathrm{R}}].timeSent, T.inp[\hat{e}_{\mathrm{R}}].tag$ to $\mathcal{A}''$, and $\mathcal{A}''$ outputs these values. Notice that during the execution, there was no 'Broadcast' event which could have output m with timestamp $timeSent$, which means that there was no such corresponding query to the OTag oracle. Thus, for these values, the OVer oracle should return $\top$ in line 6 of Alg. 25. This implies that $\mathcal{A}''$ has probability $> 2^{-|params.1^n|} + Negl(|params|)$ to win the $\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game, where n and κ are the tag length and key length, respectively, used by entities during execution of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, i.e.:

$$\Pr[\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n) = \top] > 2^{-|params.1^n|} + Negl(|params|) \tag{17}$$

Since n is the tag length and κ is the key length used by the entities during the execution of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$ and $\mathcal{A}$ polybound-satisfies $\mathcal{M}_{\mathrm{SecKeyInit}}$ using $\mathcal{X}$, then there is only negligible probability that $n \neq |params.1^n|$ or $\kappa \neq |params|$, due to model $\mathcal{M}^{\kappa \geq n}_{params} \in \mathcal{M}_{\mathrm{SecKeyInit}}$. Thus, Equation 17 must hold for κ, n where $n = |params.1^n|$ and $\kappa = |params| \geq |params.1^n|$. Then:

$$\Pr[\mathbf{Exp}^{\mathrm{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n) = \top] > 2^{-n} + Negl(\kappa)$$

for $\kappa \geq n$. Also, by Theorem 1, the total runtime of $\mathcal{A}$ is bounded by a polynomial, since 'Sec-in' is a bounded $\mathcal{X}$-operation. This implies that the length of the transcript T is bounded by a polynomial as well, and the time it takes $\mathcal{A}$' to search T should be polynomial, and thus the total runtime of $\mathcal{A}''$ should be polynomial. Therefore, since a polynomial adversary wins with more than $2^{-n} + Negl(\kappa)$ probability in the unforgeability game against MAC, then MAC is not asymptotically unforgeable (according to Def. 10 in Sec. C.5).

□

C.6.2 Proof of Claim 2

In this section, we prove Claim 2 of Theorem 2 - that $\mathsf{AuthBroadcast}$ ensures freshness and authentication assuming secure key sharing, valid n and 1^κ parameters given to the protocol, and bounded clock drift. Note that this proof is similar to the proof of Claim 2 of Theorem 2 (see Sec. C.6.1), with some changes due to the bounded clock drift assumption and the freshness requirement. The claim is restated below, for an asymptotically unforgeable message authentication scheme MAC (as defined in Appendix C.5) and $\mathcal{X} = \{\text{'Sec-in'}\}$, where 'Sec-in' is a bounded $\mathcal{X}$-operation (see Sec. 2.3 for discussion of 'Sec-in' and Sec. B.4 for discussion of bounded $\mathcal{X}$-operations).

Claim 2. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$ for $f(\Delta) = \Delta + 2\Delta_{clk}$, i.e.:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}, \mathcal{X}}_{\text{polybound}} \mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$$

for $f(\Delta) = \Delta + 2\Delta_{clk}$, where $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ is defined in Sec. C.2.2 and $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ is defined in Sec. C.3.2.

We prove Claim 2 by contradiction - namely, by showing that for $f(\Delta) = \Delta + 2\Delta_{clk}$, if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$, then MAC is not asymptotically unforgeable.

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 $\overline{\mathsf{AuthBroadcast}}$. We also define an algorithm $\mathcal{A}''$, which is used by $\mathcal{A}'$.
2. Then we show that for $f(\Delta) = \Delta + 2\Delta_{clk}$, if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{polybound}} \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ that has probability non-negligibly (in the size of input parameters) greater than 2^{-n} to 'win' against the $\overline{\mathsf{AuthBroadcast}}$ protocol (w.r.t requirement $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$), where n is a tag length parameter.
3. Lastly, we show that $\mathcal{A}''$ can 'win' the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ with probability greater than $2^{-n} + Negl(\kappa)$ for some $\kappa \geq n$, which implies that MAC is not an asymptotically unforgeable message authentication scheme.

Step 1: Protocol $\overline{\mathsf{AuthBroadcast}}$, Adversary $\mathcal{A}'$, and Subroutine $\mathcal{A}''$

$\mathcal{A}'$ takes $params \in \{0,1\}^*$ as input and works as follows:

1. $\mathcal{A}'$ modifies the code for $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ into $\overline{\mathsf{AuthBroadcast}}$. Specifically, line 4 in the 'Broadcast' function of $\mathsf{AuthBroadcast}$ (see Algorithm 23) is replaced by:

$$tag \leftarrow \overline{\mathsf{Tag}}(m \mid\mid timeSent)$$

   $\mathcal{A}'$ also changes line 3 in the 'Receive' function of $\mathsf{AuthBroadcast}$ (see Algorithm 24) to:

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

$\mathcal{A}'$ also adds the following line after line 6 in the 'Init' function of $\mathsf{AuthBroadcast}$ (see Algorithm 20):

$$\overline{\mathsf{Init}}(1^\kappa, n)$$

$\overline{\mathsf{Tag}}$ and $\overline{\mathsf{Ver}}$ are initially placeholders. When $\overline{\mathsf{Init}}(1^\kappa, n)$ is invoked, it passes the values κ and n to $\mathcal{A}'$ and starts the game $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ (see Alg. 25) if it is not already started, where $\mathcal{A}''$ is an algorithm controlled by $\mathcal{A}'$. When the game gets to line 5 (in Alg. 25), $\mathcal{A}''$ gets access to oracles OTag and OVer. It then makes $\overline{\mathsf{Tag}}$ point to OTag and $\overline{\mathsf{Ver}}$ point to OVer. Note that $\mathsf{OTag}(\cdot)$ is an oracle that takes a message m as input and returns $\mathsf{MAC}_n(k, m)$, and $\mathsf{OVer}(\cdot,\cdot)$ is an oracle that takes inputs m and tag and returns $\top$ if $\mathsf{MAC}_n(k, m) = tag$ and $\bot$ otherwise, where k is a key chosen uniformly from $\{0, 1\}^\kappa$ and used by both oracles.

2. $\mathcal{A}'$ executes $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, where $\mathcal{A}$ is discussed below. During the execution, as long as there is at least one entity, $\overline{\mathsf{AuthBroadcast}}$['Init'] is invoked and the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ is started as described above.
3. $\mathcal{A}'$ 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 and where the 'Broadcast' event happened within $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time of the event $\hat{e}_{\text{R}}$ (where Δ_{clk} is the maximum clock drift value assumed in the model $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$). In the case that $\mathcal{A}'$ finds such an event, it passes $T.inp[\hat{e}_{\text{R}}].m \mid\mid T.inp[\hat{e}_{\text{R}}].timeSent, T.inp[\hat{e}_{\text{R}}].tag$ to $\mathcal{A}''$, which outputs this to the (still ongoing) $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game; otherwise it passes a pair of randomly chosen strings $x \overset{\mathbf{R}}{\leftarrow} \{0, 1\}^n, tag \overset{\mathbf{R}}{\leftarrow} \{0, 1\}^n$, to $\mathcal{A}''$, which outputs x, tag to the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa, n)$ game.

Step 2: Adversary $\mathcal{A}$

From Definition 7, if for $f(\Delta) = \Delta{+}2\Delta_{clk}$, $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X} = \{\text{'Sec-in'}\}$ $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{polybound}} \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ such that:

$$\epsilon^{\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) > \beta^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}(params) + Negl(|params|) \tag{18}$$

where:

$$\epsilon^{\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) \equiv \Pr\left[\begin{array}{c}\pi^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}(T, params) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}})}(params)\end{array}\right]$$

and $\beta^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}(params) = 2^{-|params.1^n|}$, for $f(\Delta) = \Delta + 2\Delta_{clk}$.

We show now that Equation 18 implies Equation 19, as stated in Lemma 7 below. Note that the difference between Eq. 18 and Eq. 19 is the protocol - $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ is changed to $\overline{\mathsf{AuthBroadcast}}$.

Lemma 7. *Let $\mathcal{X} = \{\text{'Sec-in'}\}$. Suppose that there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{polybound} \mathcal{M}^{SecKeyInit}_{Drift_{\Delta_{clk}}}$ which satisfies Equation 18, for $f(\Delta) = \Delta + 2\Delta_{clk}$. Then holds:*

$$\epsilon^{\pi^{Broadcast}_{Auth_{f(\Delta)}}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}}),\mathcal{X}}(params) > \beta^{Broadcast}_{Auth_{f(\Delta)}}(params) + Negl(|params|) \tag{19}$$

Where:

$$\epsilon^{\pi^{Broadcast}_{Auth_{f(\Delta)}}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}}),\mathcal{X}}(params) \equiv \Pr\left[\begin{array}{c}\pi^{Broadcast}_{Auth_{f(\Delta)}}(T, params) = \bot, \ \ where \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)\end{array}\right]$$

and $\beta^{Broadcast}_{Auth_{f(\Delta)}}(params) = 2^{-|params.1^n|}$, for $f(\Delta) = \Delta + 2\Delta_{clk}$.

Proof of Lemma 7.

From Equation 15, $\mathcal{A}$ is a PPT adversary that 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 probability $> 2^{-|params.1^n|} + Negl(|params|)$). Recall that, according to Alg. 23, the 'Broadcast' function of the protocol outputs the message m, local time $timeSent$, and tag $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent)$, where $s.k$ is unknown to $\mathcal{A}$ due to secure key sharing (except for only negligible probability, as ensured by $\mathcal{M}_{\text{SecKeyInit}}$) and $s.n = |params.1^n|$ due to $\mathcal{M}^{\kappa \geq n}_{params} \in \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ (except for negligible probability). Also recall that, according to Alg. 24, the 'Receive' function of the protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent) = tag$ and that the local time at the receiver minus the timestamp is $\leq s.\Delta$. Since, according to $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$, local time is within Δ_{clk} of the real time, this implies that 'Receive' only accept messages sent within the last $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time (except for negligible probability). Thus, with probability $> 2^{-|params.1^n|} + Negl(|params|)$, $\mathcal{A}$ is able to output a message m, timestamp $timeSent$, and tag tag which it was not previously given such that $\mathsf{MAC}_{s.n}(s.k, m \mid\mid timeSent) = tag$ and the local time at the receiver minus $timeSent$ is $\leq s.\Delta$.

Executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$ only differ from executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}})}(params)$ by the added call to $\overline{\mathsf{Init}}$ in 'Init', which does not alter the state or the outputs of 'Init', and the changes to 'Broadcast' and 'Receive', which cause messages to be authenticated using the OTag and OVer oracles in executions of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, instead of using $\mathsf{MAC}_{s.n}(s.k, \cdot)$, where s is the state of some entity and $s.k$ is a key chosen uniformly from $\{0,1\}^{|s.1^\kappa|}$, shared securely among the entities, and unknown to $\mathcal{A}$. But $\mathsf{OTag}(\cdot)$ uses $\mathsf{MAC}_n(k,\cdot)$ to compute a tag over a message, $\mathsf{OVer}(\cdot,\cdot)$ uses $\mathsf{MAC}_n(k,\cdot)$ to verify that a tag is correct, and both oracles use the same key chosen uniformly from $\{0,1\}^\kappa$, where n is the tag length and κ is the key length used by the entities. Thus, the executions should be equivalent to $\mathcal{A}$, which implies Equation 19.

□

Completing the proof of Claim 2

We can now complete the proof of Claim 2. That is, we show that adversary $\mathcal{A}''$ wins the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa,n)$ game with probability greater than $2^{-n}+Negl(\kappa)$ for $\kappa \geq n$.

Proof. Suppose that for $f(\Delta)=\Delta+2\Delta_{clk}$, $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Auth}_{f(\Delta)}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$. Then Equation 18 holds (from Definition 7).

By Lemma 7, Equation 19 holds. This means that when $\mathcal{A}'$ runs $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, $\mathcal{A}'$ has probability $> 2^{-|params.1^n|} + Negl(|params|)$ to find that T contains an event $\hat{e}_{\text{R}}$ such that the output of $\hat{e}_{\text{R}}$ is $(\text{'}\mathsf{Receive}\text{'}, m)$, yet there is no previous 'Broadcast' event $\hat{e}_{\text{B}}$ where the input is m and where the 'Broadcast' event happened within $params.\mathcal{P}.\Delta + 2\Delta_{clk}$ real time of the event $\hat{e}_{\text{R}}$. Whenever this is the case, then $\mathcal{A}'$ gives to $\mathcal{A}''$ the values $T.inp[\hat{e}_{\text{R}}].m \;||\; T.inp[\hat{e}_{\text{R}}].timeSent, T.inp[\hat{e}_{\text{R}}].tag$, and $\mathcal{A}''$ outputs these values. Notice that during the execution, there was no 'Broadcast' event which could have output m with timestamp $timeSent$, which means that there was no such corresponding query to the OTag oracle. Thus, for these values, the OVer oracle should return $\top$ in line 6 of Alg. 25. This implies that $\mathcal{A}''$ has probability $> 2^{-|params.1^n|} + Negl(|params|)$ to win the $\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa,n)$ game, where n and κ are the tag length and key length, respectively, used by the entities during the execution of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$, i.e.:

$$\Pr[\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa,n)=\top] > 2^{-|params.1^n|} + Negl(|params|) \tag{20}$$

Since n is the tag length and κ is the key length used by the entities during the execution of $\mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\overline{\mathsf{AuthBroadcast}})}(params)$ and $\mathcal{A}$ polybound-satisfies $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$ using $\mathcal{X}$, then there is only negligible probability that $n \neq |params.1^n|$ or $\kappa \neq |params|$, due to model $\mathcal{M}^{\kappa\geq n}_{params} \in \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}}}$. Thus, Equation 20 must hold for κ, n where $n=|params.1^n|$ and $\kappa=|params| \geq |params.1^n|$. Then:

$$\Pr[\mathbf{Exp}^{\text{UF}}_{\mathcal{A}'',\mathsf{MAC}}(\kappa,n)=\top] > 2^{-n} + Negl(\kappa)$$

for $\kappa \geq n$. Also, by Theorem 1, the total runtime of $\mathcal{A}$ is bounded by a polynomial, since 'Sec-in' is a bounded $\mathcal{X}$-operation. This implies that the length of the transcript T is bounded by a polynomial as well, and the time it takes $\mathcal{A}$' to search T should be polynomial, and thus the total runtime of $\mathcal{A}''$ should be polynomial. Therefore, since a polynomial adversary wins with more than $2^{-n} + Negl(\kappa)$ probability in the unforgeability game against MAC, then MAC is not asymptotically unforgeable (according to Def. 10 in Sec. C.5).

□

C.6.3 Proof of Claim 3

In this section, we prove Claim 3 of Theorem 2 - that $\mathsf{AuthBroadcast}$ ensures correct bounded-delay delivery/receipt of broadcast messages assuming secure key sharing, valid n and 1^κ parameters given to the protocol, bounded clock drift, bounded-delay communication, and a sufficiently large freshness interval parameter given to the protocol. That this implies authenticity and freshness as well. The claim is restated below, for an asymptotically unforgeable message authentication scheme MAC (as defined in Appendix C.5) and $\mathcal{X} = \{\text{'Sec-in'}\}$, where 'Sec-in' is a bounded $\mathcal{X}$-operation (see Sec. 2.3 for discussion of 'Sec-in' and Sec. B.4 for discussion of bounded $\mathcal{X}$-operations).

Claim 3. $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ polybound-satisfies $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ using $\mathcal{X}$-operations $\{\text{'Sec-in'}\}$, i.e.:

$$\mathsf{AuthBroadcast}^{\mathsf{MAC}} \models^{\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}},\mathcal{X}}_{\text{polybound}} \mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$$

Where $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ is defined in Sec. C.2.3 and $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ is defined in Sec. C.3.3.

We prove Claim 3 by contradiction - namely, by showing that if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ using $\mathcal{X}$, then MAC does not satisfy the definition of message authentication scheme.

Proof. From Definition 7, if $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$ does not polybound-satisfy $\mathcal{R}^{\text{Broadcast}}_{\text{Receive}_{\Delta_{com}}}$ under $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ using $\mathcal{X} = \{\text{'Sec-in'}\}$, then there exists a PPT adversary $\mathcal{A} \models^{\mathcal{X}}_{\text{polybound}} \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ such that:

$$\epsilon^{\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) > Negl(|params|) \tag{21}$$

where:

$$\epsilon^{\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}}),\mathcal{X}}(params) \equiv$$

$$\Pr\left[\begin{array}{c}\pi^{\text{Broadcast}}_{\text{Receive}_\Delta}(T, params) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathsf{ACom}(\mathcal{A}),\mathsf{PCom}(\mathsf{AuthBroadcast}^{\mathsf{MAC}})}(params)\end{array}\right]$$

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

However, the model $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ 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. 16), and $\Delta_{com} + 2\Delta_{clk} \leq params.\mathcal{P}[i].\Delta$ for every entity i (see Algorithm 17, for $f(\Delta_{com}, \Delta_{clk}) = \Delta_{com} + 2\Delta_{clk}$). Note that we can assume that this is possible for an adversary $\mathcal{A}$ which satisfies $\mathcal{M}_{\text{polyAdv}}$, because the protocol's states only grow in the 'Init', 'Key-Gen', and 'Sec-in' operations, the outputs of 'Broadcast' are at most a fixed amount longer than the inputs, and $\mathcal{A}$ can control the number of 'Broadcast' events and the broadcast messages, so it can ensure that it has enough runtime left to deliver all broadcast messages.

Recall that, according to Alg. 23, the 'Broadcast' function of $\mathsf{AuthBroadcast}^{\mathsf{MAC}}$, for input message m, outputs ($\text{'}\mathsf{Broadcast}\text{'}, m, timeSent, \mathsf{MAC}_{s.n}(s.k, m \,||\, timeSent)$), where $s.n$ is the length of tags used, $timeSent$ is the local time, and $s.k$ is a key shared securely among all entities (except for negligible probability, as ensured by $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$). Also recall that, according to Alg. 24, the 'Receive' function of the $\mathsf{AuthBroadcast}$ protocol, for inputs $m, timeSent, tag$, verifies that $\mathsf{MAC}_{s.n}(s.k, m \,||\, timeSent) = tag$ and that the local time at the receiver minus the timestamp is $\leq s.\Delta$. If this holds, the 'Receive' function outputs ($\text{'}\mathsf{Receive}\text{'}, m$).

Thus, since $\mathcal{A} \models^{\mathcal{X}}_{\text{polybound}} \mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$, 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 where $(m, timeSent, \mathsf{MAC}_{s.n}(s.k, m \,||\, timeSent))$ is received as input at j within Δ_{com} real time. Since $\mathcal{M}^{\text{SecKeyInit}}_{\text{Drift}_{\Delta_{clk}},\text{Delay}_{\Delta_{com}}}$ 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 receiver's local time cannot be more than the $timeSent + params.\mathcal{P}.\Delta$. Also, by $\mathcal{M}_{\text{SecKeyInit}}$, all entities use the same tag length and the same securely shared key (except with negligible probability). Consequently, if some message is broadcast, yet the message is not successfully received, then this implies that $\mathsf{MAC}_n(k, m \,||\, timeSent) \neq \mathsf{MAC}_n(k, m \,||\, timeSent)$ (in the 'Receive' function), where n and k are the tag length and key used by the entities, respectively. This implies 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 9 in Sec. C.5, implying that MAC is not a message authentication scheme. $\square$

D Additional Specification Predicates

D.1 Examples of Models and Model Predicates

D.1.1 Faulty Entities model $\mathcal{M}^{\mathsf{F}}$: Enforce Faulty Entities Set $\perp$

We now define the $\mathcal{M}^{\mathsf{F}}$ fault model, which ensures that the adversary's output declares all the faulty entities. The base function for this model is 0, i.e., $\mathcal{M}^{\mathsf{F}} = (\pi^{\mathsf{F}}, 0)$, where we define the π^{F} predicate in Algorithm 26.

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

```
1: return (
2:     $\forall \hat{e} \in \{1, \ldots, T.e\}$ :                                   ▷ For each event
3:         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
4:         then $T.ent[\hat{e}] \in T.\mathsf{F}$                                  ▷ Then entity is in $T.\mathsf{F}$
   )
```

The π^{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 $T.\mathsf{F}$.

D.1.2 Threshold Security model $\mathcal{M}^{|\mathsf{F}| \leq f}$: up to f Faults

Next, we define the *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 27. The $\pi^{|\mathsf{F}| \leq f}$ predicate ensures that the set of faulty entities $T.\mathsf{F}$ does not exceeds the function f and uses the π^{F} predicate, to ensure that all faulty entities are indeed part of the $T.\mathsf{F}$ set.

Algorithm 27 $\pi^{|\mathsf{F}|\leq f}$ $(T,\ params)$ Predicate

```
1: return (
2:     $(|T.\mathsf{F}| \leq f\,(|T.\mathsf{N}|))$                    ▷ Max size of $T.\mathsf{F}$ is not exceeded
3:     and $\pi^{\mathsf{F}}(T, params)$                              ▷ And all faulty entities are identified
    )
```

D.1.3 $\mathcal{M}^{\mathbf{AuthCom}}_{\Delta_{com}}$: 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:

$$\pi^{\text{AuthCom}}_{\Delta_{com}}(T, params) = \pi^{\text{AuthCom-rcv}}_{\Delta_{com}}(T, params) \wedge \pi^{\text{Com}}_{\Delta_{com}}(T, params) \qquad (22)$$

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 28.

Algorithm 28 $\pi^{\text{AuthCom-rcv}}_{\Delta_{com}}$ $(T,\ params)$ Predicate

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

The $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate ensures reliable, bounded-delay delivery of messages sent. Assume that at event $\hat{e}_{\text{S}}$ of the execution, the output $out[\hat{e}_{\text{S}}]$ generated by $ent[\hat{e}_{\text{S}}]$, includes a $(\text{'send'}, m, j)$ triplet, i.e., $ent[\hat{e}_{\text{S}}]$ sends message m to $j \in \mathsf{N}$. If the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate is true for this execution, then after at most Δ_{com}, if the execution did not terminate already, then entity j would receive m from $ent[\hat{e}_{\text{S}}]$. The $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate is shown in Algorithm 29.

Algorithm 29 The $\pi^{\text{Com}}_{\Delta_{com}}$ $(T,\ params)$ Model

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

We remark that: $\pi^{\text{AuthCom}}_{\Delta_{com}}$ only applies when both sender and recipient are honest (i.e., in $\mathsf{N} - \mathsf{F}$); $\pi^{\text{AuthCom}}_{\Delta_{com}}$ only ensures delivery, sender authentication and bounded delay. This still allows receipt of duplicate messages, which may involve unbounded delay. To simplify $\pi^{\text{Com}}_{\Delta_{com}}$, we use the adversary-controlled $\tau[\cdot]$ values (line 6 of Algorithm 1). For this to be meaningful, we depend on the synchronization properties of the $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ model, discussed in Sec. 3.3.

D.1.4 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 30.

Algorithm 30 $\pi^{\text{Wake-up}}_{\Delta_{clk}}$ $(T,\ params)$ 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 $T.opr[\hat{e}'] = \text{'Wake-up'}$                ▷ And the operation is 'Wake-up'
   )
```

D.1.5 Secure shared-keys Initialization ($\mathcal{M}_{\text{SecKeyShareInit}}$) 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 31, 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. Note that this model should be assumed together with the $\mathcal{M}^{\text{Exclude}}_{\mathcal{P}[\text{'Sec-in'}]}$ model defined in Sec. C.2.1 to prevent the adversary from directly using the 'Sec-in' operation of the protocol (i.e., without using an $\mathcal{X}$-operation).

A similar model for secure (authenticated) initialization of public keys is defined in [25].

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)

Algorithm 31 $\pi_{\text{SecKeyShare}}(T,\, params)$ 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':    ▷ For each message 'Send' event and each message 'Receive' event
3:         if $T.opr[\hat{e}''] =$ 'Send'    ▷ Determine the sender and receiver of the message
           then $i_S, i_R \leftarrow T.ent[\hat{e}''], T.inp[\hat{e}''].i_R$
           else if $T.opr[\hat{e}''] =$ 'Receive'
           then $i_S, i_R \leftarrow T.inp[\hat{e}''].i_S, T.ent[\hat{e}'']$
4:         $\exists\ \hat{e}, \hat{e}'$
               s.t. $1 \leq \hat{e} < \hat{e}' < \hat{e}''$    ▷ Previously (before the 'Send' or 'Receive' event)
5:             and $T.opr[\hat{e}] =$ 'Sec-channel-setup'    ▷ Keys were securely sent
               and $T.ent[\hat{e}] = i_S$    ▷ From the sender of the message
               and $T.sec\text{-}out[\hat{e}][i_R] \neq \perp$    ▷ To the receiver of the message
6:             and $T.opr[\hat{e}'] =$ 'Sec-in'    ▷ And they were securely received
               and $T.ent[\hat{e}'] = i_R$    ▷ By the receiver of the message
               and $T.inp[\hat{e}'] = \hat{e}$
7:         and $\exists\ \hat{e}, \hat{e}'$
               s.t. $1 \leq \hat{e} < \hat{e}' < \hat{e}''$    ▷ And also previously (before the 'Send' or 'Receive' event)
8:             and $T.opr[\hat{e}] =$ 'Sec-channel-setup'    ▷ Keys were securely sent
               and $T.ent[\hat{e}] = i_R$    ▷ From the receiver of the message
               and $T.sec\text{-}out[\hat{e}][i_S] \neq \perp$    ▷ To the sender of the message
9:             and $T.opr[\hat{e}'] =$ 'Sec-in'    ▷ And securely received
               and $T.ent[\hat{e}'] = i_S$    ▷ By the sender of the message
               and $T.inp[\hat{e}'] = \hat{e}$
    )
```

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

We formalize this by assuming that each entity $i \in \mathsf{N}$ *identifies* its public key pk by outputting the pair $(\text{'}\mathsf{public\ key}\text{'}, pk) \in out[\hat{e}]$, 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.1.3; 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):

$$V_{ka}(i, pk, T) = \{\exists \hat{e} \ s.t. \ T.ent[\hat{e}] = i \ \wedge (\text{'}\mathsf{public\ key}\text{'}, pk) \in T.out[\hat{e}]\} \tag{23}$$

We now define the Verified Attribution of Statements Requirement $\mathcal{R}_{\mathcal{VAS}}$; the base-function is zero, i.e., $\mathcal{R}_{\mathcal{VAS}} = (\pi_{\mathsf{VAS}}, 0)$. 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. 23). 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 32.

Algorithm 32 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 $(\text{'signed'}, m) \in T.out[\hat{e}]$               ▷ 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 [26]. 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 accusations* requirement (and predicate).

The Non-frameability requirement and Proof of Misbehavior. The first security requirement 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 requirement is defined as $\mathcal{R}_{\mathsf{NF}} = (\pi_{\mathsf{NF}}, 0)$. The Non-frameability predicate π_{NF}, shown in Algorithm 33, returns $\bot$ if the adversary was able to output a Proof of Misbehavior for an honest entity, and $\top$ otherwise.

Algorithm 33 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 Requirement

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). The No False Accusations requirement is defined as $\mathcal{R}_{\mathsf{NFA}} = (\pi_{\mathsf{NFA}}, 0)$. To properly define the requirement predicate π_{NFA}, 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 after the requirement predicate, 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 $T.out[T.e] = (\mathrm{IA}, j, x)$, for some $j \in T.\mathsf{N}$, and both j and $T.ent[T.e]$ are honest (i.e., $j, T.ent[T.e] \in T.\mathsf{N} - T.\mathsf{F}$). See Algorithm 34. Note that we only need to consider the last event (i.e., event $T.e$), because if there is an adversary which can cause a false accusation in an earlier event, then there must be an adversary which can cause a false accusation in the last event, since the adversary can decide when to end the execution.

Algorithm 34 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 35.

Algorithm 35 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 $\mathcal{A}$ poly-satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$, then $\mathcal{A}$ poly-satisfies $\mathcal{M}$ 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. □

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 $\mathcal{A}$ poly-satisfies both $\mathcal{M}$ and $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{A}$ poly-satisfies the 'stronger' model $\widehat{\mathcal{M}} \equiv \mathcal{M} \cup \mathcal{M}'$ 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:

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

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 $\mathcal{P}$ poly-satisfies requirement $\mathcal{R}$ under $\mathcal{M}$ using the execution process operations set $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies $\mathcal{R}$ 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:

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

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:

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

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

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 $\mathcal{P}$ poly-satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies $\mathcal{R}$ 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:

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

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

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

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

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 $\mathcal{P}$ poly-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and poly-satisfies $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{P}$ poly-satisfies 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 $\mathcal{X}$, namely:

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

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:

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

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

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

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 Sec. F.1 and F.2) and their proofs (in Sec. F.3), which use the concrete security definitions from Sec. 7.2.

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 the *Model Monotonicity Lemma (concrete 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 8 (Model Monotonicity Lemma (concrete 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 $\mathcal{A}$ CS-satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$, then $\mathcal{A}$ CS-satisfies $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \mathcal{M} \tag{24}$$

We next show the *Models Union Lemma (concrete security)*, 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 9 (Models Union Lemma (concrete security)).

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

$$\left(\mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \mathcal{M} \wedge \mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{M}} \tag{25}$$

We next show the *Requirement-Model Monotonicity Lemma (concrete security)*, 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 8), which implies that the protocol satisfies the requirement for such adversaries.

Lemma 10 (Requirement-Model Monotonicity Lemma (concrete security)).

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

$$\mathcal{P} \models^{\mathcal{M}, \mathcal{X}}_{\mathsf{CS}} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}}, \mathcal{X}}_{\mathsf{CS}} \mathcal{R} \tag{26}$$

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 *Requirement Monotonicity Lemma (concrete security)* 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 11 (Requirement Monotonicity Lemma (concrete 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 $\mathcal{P}$ CS-satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ CS-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models_{\mathsf{CS}}^{\mathcal{M},\,\mathcal{X}} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models_{\mathsf{CS}}^{\mathcal{M},\,\mathcal{X}} \mathcal{R} \tag{27}$$

Finally, the *Requirements Union Lemma (concrete security)* 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 12 (Requirements Union Lemma (concrete 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 $\mathcal{P}$ CS-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and CS-satisfies $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{P}$ CS-satisfies 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 $\mathcal{X}$, namely:

$$\left(\mathcal{P} \models_{\mathsf{CS}}^{\mathcal{M},\,\mathcal{X}} \mathcal{R} \wedge \mathcal{P} \models_{\mathsf{CS}}^{\mathcal{M}',\,\mathcal{X}} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models_{\mathsf{CS}}^{\widehat{\mathcal{M}},\,\mathcal{X}} \widehat{\mathcal{R}} \tag{28}$$

F.3 Proofs of the Concrete Security Modularity Lemmas

Lemma 8 (Model Monotonicity Lemma (concrete 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 $\mathcal{A}$ CS-satisfies $\widehat{\mathcal{M}}$ using $\mathcal{X}$, then $\mathcal{A}$ CS-satisfies $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \widehat{\mathcal{M}} \Rightarrow \mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \mathcal{M} \tag{24}$$

Proof. Using Def. 4, the left side means that:

$$\left(\forall\ \mathcal{P}, params \in \{0,1\}^*, (\pi, \beta) \in \widehat{\mathcal{M}} \cup \mathcal{M}^{\mathrm{CS}}\right) :$$
$$\epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(params) \le \beta(params)$$

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

$$\Rightarrow \left(\forall\ \mathcal{P}, params \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M} \cup \mathcal{M}^{\mathrm{CS}}\right) :$$
$$\epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(params) \le \beta(params)$$

Which is the definition of $\mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \mathcal{M}$, according to Def. 4. □

Lemma 9 (Models Union Lemma (concrete security)).

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

$$\left(\mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \mathcal{M} \wedge \mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \mathcal{M}'\right) \Rightarrow \mathcal{A} \models_{\mathsf{CS}}^{\mathcal{X}} \widehat{\mathcal{M}} \tag{25}$$

Proof. By Def. 4, the left side means that:

$$\begin{gathered}\left(\forall\ \mathcal{P}, \mathit{params} \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M} \cup \mathcal{M}^{\mathrm{CS}}\right) : \\ \epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(\mathit{params}) \leq \beta(\mathit{params}) \\ \wedge \left(\forall\ \mathcal{P}, \mathit{params} \in \{0,1\}^*, (\pi, \beta) \in \mathcal{M}' \cup \mathcal{M}^{\mathrm{CS}}\right) : \\ \epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(\mathit{params}) \leq \beta(\mathit{params})\end{gathered}$$

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

$$\begin{gathered}\Rightarrow \left(\forall\ \mathcal{P}, \mathit{params} \in \{0,1\}^*, (\pi, \beta) \in \widehat{\mathcal{M}} \cup \mathcal{M}^{\mathrm{CS}}\right) : \\ \epsilon^{\pi}_{\mathsf{CS}(\mathcal{A}),\mathcal{P},\mathcal{X}}(\mathit{params}) \leq \beta(\mathit{params})\end{gathered}$$

Which is the definition of $\mathcal{A} \models^{\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{M}}$, according to Def. 4. □

Lemma 10 (Requirement-Model Monotonicity Lemma (concrete security)).

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

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{CS}} \mathcal{R} \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\,\mathcal{X}}_{\mathsf{CS}} \mathcal{R} \tag{26}$$

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

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

Since $\mathcal{M}$ is weaker than $\widehat{\mathcal{M}}$, then by Lemma 8:

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

Therefore, we have:

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

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

Lemma 11 (Requirement Monotonicity Lemma (concrete 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 $\mathcal{P}$ CS-satisfies the (stronger) requirement $\widehat{\mathcal{R}}$ under $\mathcal{M}$ using $\mathcal{X}$, then $\mathcal{P}$ CS-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$, namely:

$$\mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{R}} \Rightarrow \mathcal{P} \models^{\mathcal{M},\,\mathcal{X}}_{\mathsf{CS}} \mathcal{R} \tag{27}$$

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

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

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

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

Which is the definition of $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{CS}} \mathcal{R}$, according to Def. 5. $\square$

Lemma 12 (Requirements Union Lemma (concrete 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 $\mathcal{P}$ CS-satisfies $\mathcal{R}$ under $\mathcal{M}$ using $\mathcal{X}$ and CS-satisfies $\mathcal{R}'$ under $\mathcal{M}'$ using $\mathcal{X}$, then $\mathcal{P}$ CS-satisfies 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 $\mathcal{X}$, namely:

$$\left(\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{CS}} \mathcal{R} \wedge \mathcal{P} \models^{\mathcal{M}',\mathcal{X}}_{\mathsf{CS}} \mathcal{R}'\right) \Rightarrow \mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{R}} \tag{28}$$

Proof. Since $\widehat{\mathcal{M}}$ is stronger than $\mathcal{M}$ and stronger than $\mathcal{M}'$, then by Lemma 10:

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

Using Def. 5, this means that:

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

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

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

Which is the definition of $\mathcal{P} \models^{\widehat{\mathcal{M}},\mathcal{X}}_{\mathsf{CS}} \widehat{\mathcal{R}}$, according to Def. 5. $\square$