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.

Provable security has its roots in the seminal works rigorously proving security for constructions of cryptographic primitives, such as signature schemes [19], encryption schemes [18] and pseudorandom functions [17]. 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 [16], 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.

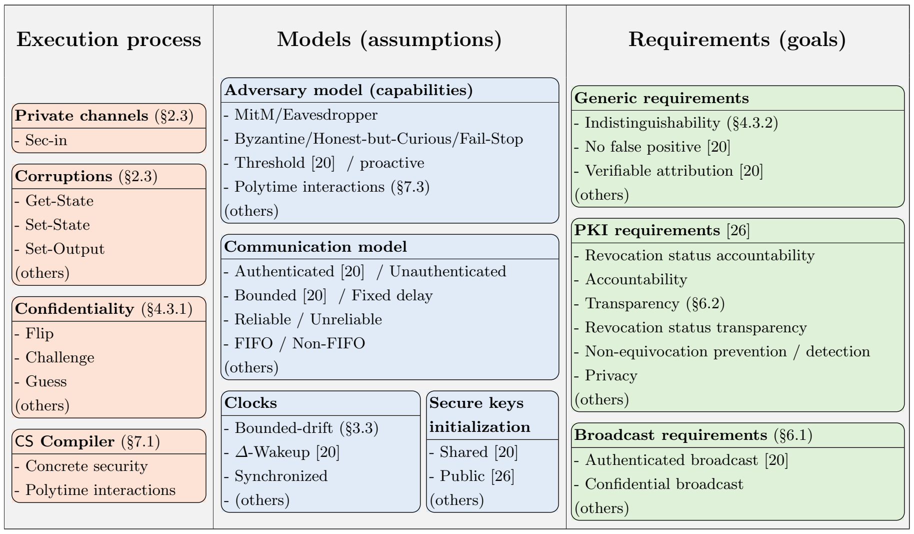


Fig. 1: The MoSS framework allows security to be specified *modularly*, i.e., 'à la carte', with respect to a set of individually-defined models (assumptions), requirements (properties/goals) and even operations of the execution process. Models, requirements and operations defined in this paper or in [20, 26] 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, 13, 22, 24, 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 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, 21, 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

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.

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 specification, 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 the full version [20], we present a simplified instance of an authenticated-broadcast protocol assuming (well-defined) bounded delay and bounded clock drift models. The *same* models are used for PKI schemes in [26].

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, 23], 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 [22]. The extendability of MoSS allows it to handle these aspects relatively simply; see Section 7.3.

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 [20].

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 [15], but there are many other proposals, most notably, Certificate Transparency (CT) [25], 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 [26] 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. Section 5 presents modularity lemmas. Section 6 shows how to apply MoSS to two different applications, a simplified authenticated broadcast protocol and PKI schemes. Section 7 describes extensions of the framework to achieve concrete security and to ensure polytime interactions. We conclude and discuss future work in Section 8.

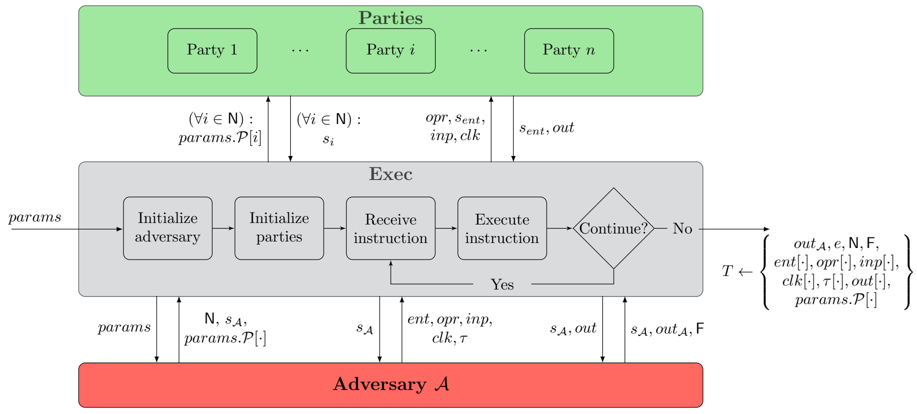
2 Execution Process

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.

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

The execution process $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$, as defined by the pseudo-code in Algorithm 1, 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.

[4] Grossly-simplified PKI ideal functionalities were studied, e.g., in [22], but without considering even basic aspects such as revocation and expiration.



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
$T \leftarrow \left\{ out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], params.\mathcal{P}[\cdot] \right\}$
$params$
$\mathsf{N}, s_{\mathcal{A}}, params.\mathcal{P}[\cdot]$
$s_{\mathcal{A}}$
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: $\forall i \in \mathsf{N}: \; s_i \leftarrow \mathcal{P}[\text{'Init'}](\perp, params.\mathcal{P}[i], \perp)$        ▷ 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 ($out_{\mathcal{A}} \neq \perp$), based on $out[e]$
11: until $out_{\mathcal{A}} \neq \perp$
12: $T \leftarrow \left(out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], params.\mathcal{P}[\cdot], s^{In}[\cdot], s^{Out}[\cdot]\right)$
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 $i \in \mathsf{N}$. The values of $params.\mathcal{P}[\cdot]$ can be restricted using *models* (see Sec. 3). 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

[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.

we did not explicitly write it, some default values are returned if the adversary specifies an operation which does not exist in the corresponding set.

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

Algorithm 2 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 $\boldsymbol{type[e] = \text{'}\mathcal{X}\text{'}}$ then        ▷ **If $\mathcal{A}$ chose to invoke an operation from $\mathcal{X}$.**
10:        $\boldsymbol{(s_{\mathcal{X}}, s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{X}[opr[e]](s_{\mathcal{X}}, s_{ent[e]}, inp[e], clk[e], ent[e])}$
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: $T \leftarrow \left(out_{\mathcal{A}}, e, \mathsf{N}, \mathsf{F}, ent[\cdot], opr[\cdot], type[\cdot], inp[\cdot], clk[\cdot], \tau[\cdot], out[\cdot], params.\mathcal{P}[\cdot], s^{Out}[\cdot], s^{In}[\cdot], sec\text{-}out[\cdot][\cdot]\right)$
18: Return $T$        ▷ Output transcript of run
```

Construction. The extended execution process (Algorithm 2) consists of the following modifications. The initialization phase (lines 1-4) has one additional line (line 3), where we initialize the 'execution operations state' $s_{\mathcal{X}}$; 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 $type[e] \in \{\text{'}\mathcal{X}\text{'}, \text{'}\mathcal{P}\text{'}\}$ 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 [20]. We define the 'Sec-in' operation in Equation 1.[6]

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

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

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.

'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 [20], 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*

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

$\mathcal{M}$, 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'*.

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.

[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.

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. For additional examples of models, see [20].

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 $\pi(T, params) = \perp$, for the transcript T of a random execution: $T \leftarrow \mathbf{Exec}^{\mathcal{X}}_{\mathcal{A},\mathcal{P}}(params)$.

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:*

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

3.2 Model-Satisfying Adversary

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

[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.

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 $\mathcal{M}^{\textbf{Drift}}_{\Delta_{clk}}$

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:    ∀ê ∈ {1, ..., T.e}:                                  ▷ For each event
3:        |T.clk[ê] − T.τ[ê]| ≤ Δ_clk                      ▷ Local clock is within Δ_clk drift from real time
4:        and if ê ≥ 2 then T.τ[ê] ≥ T.τ[ê−1]              ▷ 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 [20]).

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 $b = \bot$ then we say that *requirement predicate* π *was not satisfied* in the execution of $\mathcal{P}$, or that the *adversary won* in this execution. If $b = \top$, then we say that *requirement predicate* π *was satisfied* in this execution, or that the *adversary lost.*

We now define what it means for $\mathcal{P}$ *to 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.:*

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

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 [20] 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 $\bot$ 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. $(\text{IA}, j, x) \in T.out[T.e]$   ▷ $T.ent[T.e]$ 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., $s_{\mathcal{X}}.b \overset{\mathbf{R}}{\leftarrow} \{0, 1\}$.
- 'Challenge': executes a desired operation with *one out of two possible inputs*, according to the value of $s_{\mathcal{X}}.b$. Namely, when $\mathcal{A}$ outputs $opr[e] =$ 'Challenge', the execution process invokes:

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

  where $inp[e].opr \in \mathcal{P}$ (one of the operations in $\mathcal{P}$) and $inp[e].inp$ is an 'array' with two possible inputs, of which only one is randomly chosen via $s_{\mathcal{X}}.b$, hence, the $inp[e].inp[s_{\mathcal{X}}.b]$ notation.
- 'Guess': checks if a 'guess bit', which is provided by the adversary as input, is equal to $s_{\mathcal{X}}.b$, and returns the result in *sec-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 $\mathcal{R}^{\pi_{\mathsf{MsgConf}}}_{\text{IND}}$

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 $\mathcal{R}^{\pi_{\mathsf{MsgConf}}}_{\text{IND}} = (\text{IND}^{\pi_{\mathsf{MsgConf}}}, \frac{1}{2})$, 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.

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

Algorithm 6 π_{MsgConf} $(T,\ params)$ Predicate

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

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 [20]. The concrete security lemmas and their proofs are in [20].

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:

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

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:

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

6 Using MoSS for Applied Specifications

In this section, we give a taste of how MoSS can be used to define applied security specifications, with realistic, non-trivial models and requirements. In Section 6.1, we discuss AuthBroadcast, a simple authenticated broadcasting protocol, which we use to demonstrate the use of MoSS's modularity lemmas. In Section 6.2 we discuss PKI schemes, which underlie the security of countless real-world applications, and show how MoSS enables rigorous requirements and models for PKI schemes. The definitions we show are only examples from [26], 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 [20], 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:

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 [14], 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: $\mathcal{M}^{Drift}_{\Delta_{clk}}$*.* [26] 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 [26].

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

$$\pi_{\Delta\mathsf{TRA}}(T, params) \equiv \begin{bmatrix} (\psi, \rho, pk, \iota, \iota_M) \leftarrow T.out_{\mathcal{A}} : \\ \textsc{HonestEntity}(T, params, \iota) \wedge \\ \textsc{CorrectPublicKey}(T, params, \iota, pk, \rho.\iota) \wedge \\ \textsc{ValidCertificateAttestation}(T, params, \{\Delta\mathsf{TRA}\}, \psi, pk, \rho) \wedge \\ \textsc{HonestEntity}(T, params, \iota_M) \wedge \\ \textsc{IsMonitor}(T, params, \iota_M, \rho.\iota) \wedge \\ \textsc{HonestMonitorUnawareOfCertificate}(T, params) \wedge \\ \textsc{WasNotAccused}(T, params) \end{bmatrix}$$

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) - 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.

[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.

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 $\mathcal{A}$.

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.

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,

[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 $\mathcal{A}$.

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
    )
```

(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 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 $\mathcal{A} \overset{\mathcal{X}}{\models}_{\text{CS}} \mathcal{M}$, 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.:*

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

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* $\mathcal{P} \models^{\mathcal{M},\mathcal{X}}_{\mathsf{CS}} \mathcal{R}$*, 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.:*

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

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 $\textbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$. 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 $\textbf{Exec}^{\mathcal{X}}_{\mathsf{CS}(\mathcal{A}),\mathcal{P}}(params)$, 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 total runtime is polynomial even if we allow linear growth in state and outputs. We can focus on such 'reasonable' protocols by including an appropriate requirement in the specifications. The technical details are omitted.

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

[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.

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.

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., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: CRYPTO (2015)
14. 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`
15. CCITT, B.B.: Recommendations X. 509 and ISO 9594-8. Information Processing Systems-OSI-The Directory Authentication Framework (Geneva: CCITT) (1988)
16. Degabriele, J.P., Paterson, K., Watson, G.: Provable security in the real world. IEEE Security & Privacy **9**(3), 33–41 (2010)
17. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM **33**(4), 792–807 (Oct 1986)
18. Goldwasser, S., Micali, S.: Probabilistic Encryption. Journal of Computer and System Sciences **28**(2), 270–299 (1984)
19. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on computing (1988)
20. 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`
21. 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`
22. Hofheinz, D., Shoup, V.: Gnuc: A new universal composability framework. Journal of Cryptology **28**(3), 423–508 (2015)
23. Hofheinz, D., Unruh, D., Müller-Quade, J.: Polynomial runtime and composability. Journal of Cryptology **26**(3), 375–441 (2013)
24. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology pp. 1–124 (2020)
25. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Jun 2013). https://doi.org/10.17487/RFC6962
26. Leibowitz, H., Herzberg, A., Syta, E.: Provable security for PKI schemes. Cryptology ePrint Archive, Report 2019/807 (2019), `https://eprint.iacr.org/2019/807`
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)