The Modular Specifications Security Framework

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

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

Abstract. We present the *Modular Specifications Security (MoSS) framework*, where security specifications are defined with respect to a specific *model predicate* $\mathcal{M}$. This allows analysis of even complex schemes and protocols, e.g., PKI schemes, under well-defined adversary, communication and synchronization models, in a modular and flexible way, and allows to analyze such schemes in both simplified and realistic models. The framework facilitates reuse of definitions, and, indeed, several of the model predicates and security specifications we define, seem 'generic' and reusable in analysis of other practical protocols.

1 Introduction

Cryptographic protocols are traditionally designed and analyzed under different assumptions, where the expected outcome is to achieve provable security. Most commonly, cryptographic protocols are specified and analyzed under specific adversary models. Other assumptions, related to communication and synchronization models are not always relevant, but when they are, highly simplified models are often applied, e.g., perfect synchronization ('rounds model'). Understandably, since often there are no (relatively) simple tools that can be used to achieve stronger evaluation against more realistic models, such simplifications are used. However, this should serve as a motivation towards developing better evaluation frameworks, and should not be considered as 'good enough'.

In particular, it is an established challenge to apply provable security on to applied cryptographic protocols, especially due to the realistic communication and synchronization models which such protocols operate under. A motivational example of such schemes are PKI schemes, which these aspects have crucial implications on their security. When considering the use of existing rigorous frameworks for analysis of distributed algorithms, e.g., the I/O automata [3] or reactive systems [5], the results are usually overly complex, even when focusing on relatively simple algorithms and requirements; applying them to complex tasks, such as PKI schemes and to cryptographic protocols, appears prohibitively challenging.

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

Instead, we present the *Modular Specifications Security (MoSS) framework*, which is a significant (positive) deviation from the 'classical' approach of defining security using games (experiments), with a given protocol and adversary. Namely, while in the 'classical' game-based approach, the model is an integral part of the game/experiment, in our framework, the model $\mathcal{M}$ is a well-defined 'standalone' PPT algorithm, *separated* from the actual experiment. This approach taken by the MoSS framework, yields an important result. The MoSS framework allows a clear and concise method to take any well-defined security specification ξ and formally prove whether a given protocol $\mathcal{P}$ *satisfies* specification ξ (with negligible advantage or with advantage ϵ) *under model* $\mathcal{M}$.

The *Modular Specifications Security (MoSS) framework* cleanly separates the specification and analysis of security into three components: the *execution process*, *model* and *specifications*. The first component is a well-defined, and relatively straightforward, *adversary-driven execution process* (Algorithm 1). This execution process defines precisely the process of *executing a protocol* $\mathcal{P}$ *under adversary* $\mathcal{A}$, giving the adversary extensive control over the operation of the environment, including communication, local-clock values, inputs from the application, and faults. (The execution process only imposes some basic limitations, e.g., events are ordered, rather than potentially only partially-ordered, as allowed in [3, 5].)

The second component of the framework is the definition of *models*, where a model $\mathcal{M}$ is a pair consisting of a *model predicate* $\mathcal{M}.\pi$ and a *base function* $\mathcal{M}.\beta$. A model predicate $\mathcal{M}.\pi$ classifies the execution of a protocol $\mathcal{P}$ under adversary $\mathcal{A}$ as 'valid' ($\top$) or 'invalid' ($\bot$), effectively enforcing one or multiple restrictions on the adversarial control of the execution, including initialization assumptions, limitations on number and type of faults, maximal delay and/or maximal clock drift. Furthermore, a model predicate may enforce restrictions on the use of resources by the adversary, such as numbers of queries or running time. The base function specifies the 'base' probability that a model predicate $\mathcal{M}.\pi$ outputs $\bot$. The sum of the base function $\mathcal{M}.\beta$ and the *advantage function* ϵ (or in the asymptotic definitions, $\mathcal{M}.\beta$ and a polynomial) limits the maximum probability that the predicate outputs $\bot$ for any given parameters p.

The third component of the framework is the definition of *specifications*, which are similar to models except that they are used to define what is provided *by* a protocol or scheme, given some model. A specification ξ is a pair, $\xi = (\xi.\pi, \xi.\beta)$, where the specification predicate $\xi.\pi$ and the base function $\xi.\beta$ are used in the same way as model predicates and base functions.

We believe that this approach is not only likely to be beneficial for many cryptographic protocols and distributed systems, but would also make it easier to compare models and specifications between different works. Furthermore, we expect the separation between the model aspects and the problem-specific aspects of security definitions to allow cleaner definitions, as well as to allow the reuse of well-defined models $\mathcal{M}$.

Organization. Section 2 presents the basic integrity and availability (IA) execution model of the framework and asymptotic definitions for models and

specifications. Section 3 discusses the CIA execution process, which is an extension of the IA execution process which additionally allows confidentiality (and other indistinguishability-based) models and specifications. Section 4 extends the framework to allow concrete security (rather than asymptotic) definitions for models and specifications. Section 5 provides further model and specification predicates. We conclude and discuss future work in Section 6.

2 The MoSS Framework Building Blocks

Traditionally, research onto cryptographic schemes and protocols tend to focus on the adversary's capabilities and whether some properties can be achieved under a specific adversary model or not. This is acceptable, when the analyzed system is not concerned with other assumptions. For example, when defining the security of encryption schemes or zero-knowledge proofs, there is no significance to the properties of the communication between the parties, or to the synchronization of their clocks. However, communication and synchronization assumptions are crucial factors in the operation and security of distributed systems. For instance, in PKI schemes, some of the properties are *time related*, and as such, the scheme's security strongly depends on the communication and synchronization assumptions. Therefore, there is a clear need to bootstrap the communication and synchronization assumptions into the analysis of such systems; otherwise, analysis that overlooks them completely misses its purpose, and it is therefore, incomplete.

One way to address this challenge would be to adopt a specific, simple model for communication and synchronization, e.g., the synchronous 'rounds model', where all parties operate in lockstep, round by round, and messages sent at round i are delivered in round $i + 1$. Using such a fixed model has the benefit of making it easy to focus on the cryptographic aspects, e.g., present rigorous definitions and prove security (by reductions). However, real systems are more complex; clocks are not fully synchronized, and communication is rarely, if ever, perfectly synchronous. This creates a key dilemma in the design and analysis of cryptographic protocols; should the design and analysis assume a simplified model and focus on the cryptographic aspects of the protocol, or should it use a more realistic model, one that takes into account communication and clock synchronization and allows a *genuine* analysis of practical systems?

The reader might consider this dilemma as a 'no-brainer'; obviously, an analysis should always be with respect to a genuine (as possible) model. Unfortunately, in many cases, it is extremely non-trivial, and as a result, many designs and analyses do choose to focus on the cryptographic, ignoring some key-aspects along the way. We now details how the MoSS framework solves that by splitting the design into three components: a generic *adversary-driven execution process*, *models*, and security *specifications*. We begin with how the framework's basic execution process works.

2.1 Exec_IA: An Adversary-Driven Execution Process

We now present the adversary-driven execution process $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(1^\kappa)$ as defined by the pseudo-code in Algorithm 1. The execution process specifies the details of running a given protocol $\mathcal{P}$ with a given adversary $\mathcal{A}$, both modeled as efficient (PPT) functions. The execution process does not enforce any assumptions or model on the communication, synchronization, or inputs to the protocol; all of that is controlled entirely by the adversary $\mathcal{A}$. Furthermore, the execution allows the adversary to set and retrieve the state of any entity as well as messages sent and received, allowing for different failure models.

Notation. To allow the 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, execution of an operation defined by $\mathcal{P}$ over entity i, is expressed using the following notation: $\mathcal{P}(s, opr, inp, clk)$, where s is the *local state* of entity i, opr identifies the specific 'operation' or 'function' to be invoked, inp is the set of inputs to that operation/function, and clk is the current value of the local clock of entity i. The output of such execution is a tuple $(s, out, sec\text{-}out)$, where s is the state of entity i *after* the operation's execution, out is the regular output of the executed function, and $sec\text{-}out$ is 'secure output' - namely, $sec\text{-}out$ contains values to be shared with entities through a secure channel, which is not accessible to $\mathcal{A}$.

The execution process reserves three specific strings, 'Set-state', 'Set-output' and 'Get-state', to define *generic operations*, which are *not* part of $\mathcal{P}$. We use these reserved operations to express specific adversarial capabilities which are allowed by some adversarial models. Namely, an adversary can expose the local state ('Get-state') of an entity, e.g., corresponding to a 'honest but curious' adversary; an adversary can control the state ('Set-state') of an entity; and an adversary can control the output ('Set-output'), including the secure output, of an entity, e.g., corresponding to a 'malicious' adversary ('byzantine' fault model).

Additionally, we assume that $\mathcal{P}$ includes an operation 'Sec-in'. The execution process uses this operation to *securely* give values to an entity. Specifically, an adversary can set the operation to 'Sec-in' and choose some previous secure output (without seeing it - i.e., $\mathcal{A}$ can only choose the event *during which* it was output), which will be given as input to $\mathcal{P}$. This provides a way to assume that certain information is securely shared, which may be used, e.g., if anonymous or confidential channels must be assumed without providing appropriate mechanisms at the protocol level.

We use the *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 execution events within the execution process. Note, however, that e does not represent any clock value and is independent of any clock synchronization model, and specifically, is not controlled by the adversary. Rather, e allows us to index a sequence of operations performed within one execution loop, and precisely refer to each event of the execution. The local clock clk and real time clock τ are used to represent and model different clock assumptions. 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. The execution process does not place any restrictions on these values and enforcing any constraints is left to the appropriate clock synchronization model. The specific clock synchronization model provided in this work ensures that $\tau[e]$ is monotonously increasing and uses this property to enforce other communication and clock properties (see Sections 5.1.3 and 5.1.4).

Construction. We now discuss the three main components of the execution process (Algorithm 1), that is, the initialization, main loop execution and termination.

Initialization. In line 1, we allow the adversary to set its state and to choose the set of entities N. In line 2, we set the initial state for each entity i by invoking the 'Init' operation of the protocol $\mathcal{P}$ with inputs (i, p); in other words, each entity receives its identifier i and the security parameters p and performs its initialization operation. 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 4-14).

Main execution loop (lines 4-14). The design allows the adversary $\mathcal{A}$ to have a generous control over the execution. Specifically, in each event e, $\mathcal{A}$ determines (line 6) an operation $opr[e]$ to be applied to an entity $ent[e] \in \mathsf{N}$, with input $inp[e]$ and its local clock value $clk[e]$. Additionally, the adversary selects the global, real time clock value set to $\tau[e]$.

After the adversary defines the specific operation and inputs (line 6), the event is executed (lines 7-11). There are five options for each event specified as an operation $opr[e]$. More concretely, if $\mathcal{A}$ sets $opr[e]$ = 'Set-state' (line 7), then the state $s_{ent[e]}$ of entity $ent[e]$ is set to $inp[e]$; if $\mathcal{A}$ sets $opr[e]$ = 'Set-output' (line 8) then the outputs $out[e], sec\text{-}out[e][\cdot]$ are set to values chosen by $\mathcal{A}$ as $inp[e]$; and if $\mathcal{A}$ sets $opr[e]$ = 'Get-state' (line 9), then the output of the state $s_{ent[e]}$ of entity $ent[e]$, including any private state, e.g., private keys, is exposed to $\mathcal{A}$ and saved as $out[e]$.

Alternatively, if $\mathcal{A}$ sets $opr[e]$ = 'Sec-in' (line 10), then entity $ent[e]$ can receive values securely from another entity. This is done through the *sec-out* mechanism. Specifically, $\mathcal{A}$ chooses some event $\hat{e} = inp[e]$, and the protocol receives $ent[\hat{e}], sec\text{-}out[\hat{e}][ent[e]]$ as input, which is the identifier of the 'sender' ($ent[\hat{e}]$) along with the value(s) that were *securely* sent in event $\hat{e}$ to entity $ent[e]$. Note that $\mathcal{A}$ is never given the contents of *sec-out* during the execution process.

Otherwise, if $opr[e]$ is set to any other operation (line 11), then the process invokes the protocol $\mathcal{P}$ specific function over the state $s_{ent[e]}$ of entity $ent[e]$, with inputs $opr[e], inp[e], clk[e]$. This results in a new state $s_{ent[e]}$ and output $out[e]$ for entity $ent[e]$. The execution process allows the adversary to set and get the state of any entity; however, a specific execution model $\mathcal{M}$ may forbid such operations, e.g., return $\perp$ for executions where the adversary performs them. In line 13, the adversary processes the output $out[\hat{e}]$ of the operation $opr[\hat{e}]$. The adversary may modify its state $s_{\mathcal{A}}$, and outputs a value $out_{\mathcal{A}}$; when $out_{\mathcal{A}} \neq \perp$, the execution moves to the termination phase; otherwise the loop continues.

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

$(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}(p)$ ▷ *Initialize adversary*

$\forall i \in \mathsf{N}: \ s_i \leftarrow \mathcal{P}(\perp, \text{'Init'}, (i, p), \perp)$ ▷ *Initialize entities' local state*

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

repeat

  $e \leftarrow e + 1$ ▷ *Advance loop's counter*

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

  if $opr[e]$ ='Set-state' **then** ▷ *When $\mathcal{A}$ wants to change the local state of entity $ent[e]$*

    $s_{ent[e]}, out[e], sec\text{-}out[e][\cdot] \leftarrow inp[e], \perp, \perp$

  else if $opr[e]$ ='Set-output' **then** ▷ *When $\mathcal{A}$ wants to change the output of entity $ent[e]$*

    $out[e], sec\text{-}out[e][\cdot] \leftarrow inp[e]$

  else if $opr[e]$ ='Get-state' **then** ▷ *When $\mathcal{A}$ wants to get the current local state of entity $ent[e]$*

    $out[e], sec\text{-}out[e][\cdot] \leftarrow s_{ent[e]}, \perp$

  else if $opr[e]$ ='Sec-in' **then** ▷ *When $\mathcal{A}$ wants entity $ent[e]$ to receive secure output*
  ▷ *$\mathcal{A}$ specifies a previous event in $inp[e]$ and then the process uses $(ent[inp[e]], sec\text{-}out[inp[e]][ent[e]])$ as input to $\mathcal{P}$*

    $(s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}(s_{ent[e]}, \text{'Sec-in'}, (ent[inp[e]], sec\text{-}out[inp[e]][ent[e]]), clk[e])$

    ▷ *$\mathcal{P}$ returns the state of entity $ent[e]$, the output, and the secure output*

  else ▷ *Otherwise, $\mathcal{A}$ wants to execute operation $opr[e]$ of $\mathcal{P}$ with input $inp[e]$ over entity $ent[e]$ with local clock of $clk[e]$*

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

  end if

  $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}(s_{\mathcal{A}}, out[e])$ ▷ *Inform $\mathcal{A}$ of the value of $out[e]$ and allow $\mathcal{A}$ to decide whether to continue ($out_{\mathcal{A}} = \perp$), or to terminate the loop ($out_{\mathcal{A}} \neq \perp$)*

until $out_{\mathcal{A}} \neq \perp$

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

Return T

Termination (lines 15-16). Upon termination, the process returns the *execution transcript* T (line 16), containing the relevant values from the execution. Namely, T contains the values of $ent[\hat{e}], opr[\hat{e}], inp[\hat{e}], clk[\hat{e}], \tau[\hat{e}], out[\hat{e}]$ and $sec\text{-}out[\hat{e}][\cdot]$ for each of the events $\hat{e} \leq e$, where e is the index of the last event, as well as 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 13). Private values such as entities' private keys are not part of the execution transcript unless provided as $inp[\hat{e}]$ for $opr[\hat{e}]$ = 'Set-state' or properly extracted using $opr[\hat{e}]$ = 'Get'. We allow $\mathcal{A}$ to output F, so an adversary model can specify which entities are included in F (i.e., which entities are considered 'faulty').

Limitations. The execution process supports a large variety of models. For example, the adversary may control the state ('Set-state'), learn the state ('Get-state'), and directly control the output ('Set-output') of every party $ent[\hat{e}] \in \mathsf{N}$, but this may be restricted (or fully prohibited) by specific model $\mathcal{M}$, allowing different fault models (honest-but curious, threshold, adaptive, proactive, etc.). However, **Exec_IA** allows the adversary to control all inputs events, while typical definitions of confidentiality and indistinguishability specifications provide randomized inputs which are not directly observable by the adversary. To allow such definitions, we extend the **Exec_IA** process to the **Exec_CIA** process in section 3.

2.2 Specifications

To describe both the model *assumed by* a protocol or scheme (i.e., different adversary, communication, clock-synchronization assumptions, etc.) and the property/ies (relating to security, communication, etc.) *provided by* the protocol or scheme, we define *specifications* and *advantage functions*. Specifications are used to describe both the assumptions (referred to as the models) and the goals for protocols or schemes. We define a specification as a pair containing the *specification predicate* and the *base function*. Namely, a specification ξ is:

$$\xi = (\xi.\pi, \xi.\beta) \tag{1}$$

where $\xi.\pi$ is the specification predicate and $\xi.\beta$ is the base function.

A specification predicate takes an execution transcript T and a parameter p as input, and it returns $\top$ if T satisfies the predicate π for the given value of p and $\bot$ otherwise. (In the asymptotic definitions, the parameter p is a single unary value which we refer to as the *security parameter*.)

The base function gives the 'base' probability of success for an adversary; for example, for collision- or forgery-preventing specifications, the base function may be 2^{-l}, where l is the output block size of a hash function or a key length; for indistinguishability-based specifications (see section 3), the base function may be $\frac{1}{2}$ (a constant).

An advantage function ϵ is the maximum *advantage* probability that the adversary may have (under the specifications), *over* the 'base' probability. Namely, intuitively, a specification is *satisfied with advantage* ϵ as long as the adversary's

success probability is below $\xi.\beta(p) + \epsilon(p)$. The sum $\xi.\beta(p) + \epsilon(p)$ for any given p should be in the range $[0, 1]$. However, for the asymptotic definitions, we do not use specific functions ϵ but rather say that a specification is *satisfied with negligible advantage* (definitions are given in the next subsections).

As mentioned above, specifications are used to describe both the assumptions and the goals of protocols or schemes - namely, there are two 'types' of specifications. The first are model specifications (referred to as *models*), which describe assumptions about adversary capabilities, clocks, communication channels, etc. The second are specifications which describe the qualities provided by the protocol or scheme. Note that specifications can easily be combined or defined in terms of each other; e.g., multiple sub-model predicates may be combined into a model predicate which is the conjunction of the sub-model predicates.

In the following subsections, we further describe specifications. First, we focus on models. We define what it means for an adversary to IA-satisfy a model with negligible advantage, and we give an example of a model predicate, namely, the predicate $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$. Later, we define what it means for a protocol to IA-satisfy a specification with negligible advantage under some model, and we give an example of a specification predicate, namely, the predicate AuthComRcv.

2.3 Models

The execution process described in subsection 2.1 specifies the details of running a protocol $\mathcal{P}$ against an adversary $\mathcal{A}$; however, it does not, on its own, *restrict* the adversary, nor it considers the *adversary model*, i.e., the exact capabilities of the adversary. Furthermore, other assumptions, i.e., communication and synchronization assumptions are also decoupled from the execution process. The rationale behind this design is that the security of distributed systems and cryptographic protocols should be stated and analyzed with respect to a specific *model*. Therefore, by separating the execution process from the model, this separation allows to take a transcript T of any execution of protocol $\mathcal{P}$ with adversary $\mathcal{A}$, and examine whether transcript T satisfies the predicate $\mathcal{M}.\pi$ of a specific model $\mathcal{M}$. By the term *model*, we mean a combination of *adversary capabilities*, *communication assumptions* and *clock-synchronization assumptions*:

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

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

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

This section is structured as follows. We begin with the definitions of what it means to *IA-satisfy a model $\mathcal{M}$ with negligible advantage* in §2.3.1. Then, in §5.1.3, we give the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ predicate as an example. In §5.1.1, §5.1.2, §5.1.3, and §5.1.4, we present additional examples of model predicates representing various assumptions.

2.3.1 Model Satisfying Adversary

Models are used to restrict the capabilities of the adversary as well as the events that can happen in the execution process. This includes limiting of the possible faults, defining initialization assumptions, and defining the communication and synchronization models. Hence, we ensure that a given adversary $\mathcal{A}$ followed a given model $\mathcal{M}$'s restrictions in a given execution transcript T by applying $\mathcal{M}.\pi$ to T and p, where p is a unary value $p = 1^\kappa$ for some $\kappa \in \mathbb{N}$. We say that $\mathcal{A}$ *IA-satisfies $\mathcal{M}$ with negligible advantage* if for every constant c, there is some $\kappa' \in \mathbb{N}$ such that the probability that $\mathcal{M}.\pi(T, 1^\kappa)$ returns $\perp$ is less than $\mathcal{M}.\beta(1^\kappa) + \kappa^c$ for all $\kappa > \kappa'$, as defined next.

Definition 1 (Adversary $\mathcal{A}$ IA-satisfies model $\mathcal{M}$ with negligible advantage). *Let $\mathcal{A}$ be a PPT algorithm, and let $\mathcal{M} = (\mathcal{M}.\pi, \mathcal{M}.\beta)$ be a model, where $\mathcal{M}.\pi$ is a predicate (i.e., outputs $\top$ or $\perp$) and $\mathcal{M}.\beta$ is a function $\mathcal{M}.\beta(1^\kappa) : \{1\}^* \to [0, 1]$. We say that* adversary $\mathcal{A}$ IA-satisfies model $\mathcal{M}$ with negligible advantage, *denoted as $\mathcal{A} \models_{poly} \mathcal{M}$, if for every protocol $\mathcal{P}$ and every constant c, there is some $\kappa' \in \mathbb{N}$ such that for all $\kappa > \kappa'$, the probability that the model predicate $\mathcal{M}.\pi$, applied to a random resulting* **Exec_IA** *execution transcript T and security parameter 1^κ, is not satisfied, is less than $\mathcal{M}.\beta(1^\kappa) + \kappa^c$,*

$$\mathcal{A} \models_{poly} \mathcal{M} \stackrel{def}{=} \forall\ \mathcal{P}, c, \exists\ \kappa' \in \mathbb{N}\ s.t.\ \forall\ \kappa > \kappa' : \Pr\left[\begin{array}{l}\mathcal{M}.\pi\,(T, 1^\kappa) = \perp\ : \\ T \leftarrow \textbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(1^\kappa)\end{array}\right] < \mathcal{M}.\beta(1^\kappa) + \kappa^c \tag{2}$$

2.3.2 $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$: bounded-delay communication model predicate

As an example of a model predicate, we present the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate, which ensures reliable, bounded-delay delivery of messages. (Further model predicates can be found in section 5.) The base function used with the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate would usually be $\beta(1^\kappa) = 0$, since an adversary should have no significant advantage to prevent the timely delivery of messages (regardless of the value of the security parameter).

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

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

The $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ model predicate ensures reliable, bounded-delay delivery of messages sent. Assume that at event $\hat{e}_{\mathrm{S}}$ of the execution, the output $out[\hat{e}_{\mathrm{S}}]$ generated by $ent[\hat{e}_{\mathrm{S}}]$, includes a $(\text{'send'}, m, j)$ triplet, i.e., $ent[\hat{e}_{\mathrm{S}}]$ sends message m to $j \in \mathsf{N}$. If the $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ model predicate is true for this execution, then after at most Δ_{com}, if the execution did not terminate already, then entity j would receive m from $ent[\hat{e}_{\mathrm{S}}]$. The $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ model predicate is shown in Algorithm 11.

Algorithm 2 $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ $(T,\, p)$ Predicate

```
1:  return (
2:    ∀ê_S  s.t  1 ≤ ê_S < T.e:
3:        if  ( ∃('send', m, i_R) ∈ T.out[ê_S]      ▷ If the output includes a send triple
4:            and T.τ[T.e] ≥ T.τ[ê_S] + Δ_com       ▷ And execution did not terminate yet after Δ_com real time
5:            and T.ent[ê_S] ∈ T.N − T.F )          ▷ And the entity is honest
6:        then ∃ê_R > ê_S                           ▷ Then there is a later event
7:            s.t.  T.τ[ê_S] + Δ_com ≥ T.τ[ê_R]     ▷ Within Δ_com real time
8:            and T.ent[ê_R] = i_R                  ▷ Where the entity is the intended recipient in the send triple
9:            and T.opr[ê_R] = 'Receive'            ▷ And which is a receive event
10:           and T.inp[ê_R] = (m, T.ent[ê_S])      ▷ And in which the entity receives the message from the sender
      )
```

We remark that $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ only applies when both sender and recipient are honest (i.e., in $\mathsf{N} - \mathsf{F}$), and $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$ only ensures delivery and bounded delay. This still allows receipt of duplicate messages, which may involve unbounded delay, and receipt of messages that were never sent. To simplify $\mathcal{M}^{\mathrm{Com}}_{\Delta_{com}}$, we use the adversary-controlled $\tau[\cdot]$ values (line 6 of Algorithm 1).

2.4 Security Specifications

The execution process and the model facilitate precise definition of security specifications, using predicates ('experiments') over the results of the execution. This is separate from the validation of the model, allowing modular specification of security specifications. Namely, different works may reuse the same security specifications (and execution process) but use other, possibly more realistic (and more complex) models, expressing different adversary capabilities, restrictions on usage, and assumptions on communication and synchronization. Similarly, different works may reuse the same models to study additional security specifications. The separation between the definition of the model and of the specifications also allows definition of *generic specification predicates.* Generic specification predicates are applicable to different protocols and problems. We identify four generic specification predicates, that appear relevant to many security protocols. These specification predicates focus on attributes of messages, i.e., non-repudiation,

and on detection of misbehaving entities (see §5.1.7). This approach is quite different from the current way of defining security for cryptographic schemes and protocols; it takes some time and effort to get used to the separate model and security definitions. However, we found that with a little use, the advantages become clear and the approach becomes natural and convenient, facilitating modularity and reuse of specification predicates and model predicates, and allowing for proper comparison of security guarantees between different schemes.

2.4.1 Model-Secure Specifications

To complete the presentation of the execution process, we now discuss how it is used to define specific security specifications and properties and to analyze whether these properties are ensured by a given protocol $\mathcal{P}$, under given model $\mathcal{M}$, interacting with any PPT adversary $\mathcal{A}$. While we present *game/experiment*-based definitions, future work may consider other forms of definitions, such as simulation-based.

A protocol $\mathcal{P}$ would typically have multiple security properties, i.e., satisfy multiple security specifications. We define a *security specification* ξ as a pair $(\xi.\pi, \xi.\beta)$, where $\xi.\pi$ a predicate and $\xi.\beta$ is the base function. Again, the parameter p is a unary security parameter, $p = 1^\kappa$. Let b be the outcome of $\xi.\pi$ applied to $(T, 1^\kappa)$, where T is the transcript of the execution process $(T = \mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(1^\kappa))$ and p are the parameters, i.e., $b \leftarrow \xi.\pi(T, 1^\kappa)$; if $b = \perp$ then we say that *specification predicate $\xi.\pi$ was not satisfied* in this execution of $\mathcal{P}$, or that the *adversary won* in this execution; and if $b = \top$, then we say that *specification predicate $\xi.\pi$ was satisfied* in this execution, or that the *adversary lost*. We say that *$\mathcal{P}$ IA-satisfies ξ with negligible advantage under model $\mathcal{M}$* if for every PPT adversary $\mathcal{A}$ which IA-satisfies $\mathcal{M}$ with negligible advantage, for every constant c, there is some $\kappa' \in \mathbb{N}$ such that the probability that $\xi.\pi(T, 1^\kappa)$ returns $\perp$, where $T \leftarrow \mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(1^\kappa)$, is less than $\xi.\beta(1^\kappa) + \kappa^c$ for all $\kappa > \kappa'$, as given in the next definition.

Definition 2 (Protocol $\mathcal{P}$ IA-satisfies specification ξ with negligible advantage under model $\mathcal{M}$). *Let $\xi = (\xi.\pi, \xi.\beta)$ be a specification, where $\xi.\pi$ is a predicate and $\xi.\beta$ is a function $\xi.\beta(1^\kappa) : \{1\}^* \to [0,1]$. We say that protocol $\mathcal{P}$ IA-satisfies specification ξ under model $\mathcal{M}$, denoted $\mathcal{P} \models^{\mathcal{M}}_{poly} \xi$, if for every PPT adversary $\mathcal{A}$ that IA-satisfies $\mathcal{M}$ with negligible advantage, for every constant c, there is some $\kappa' \in \mathbb{N}$ such that for all $\kappa > \kappa'$, the probability that the specification predicate $\xi.\pi$, applied to a random resulting* **Exec_IA** *execution transcript T and security parameter 1^κ, is not satisfied, is less than $\xi.\beta(1^\kappa) + \kappa^c$, i.e.:*

$$\mathcal{P} \models^{\mathcal{M}}_{poly} \xi \stackrel{def}{=} \forall\, \mathcal{A} \in PPT, c,\ \text{if } \mathcal{A} \models_{poly} \mathcal{M} \text{ then } \exists\, \kappa' \in \mathbb{N} \text{ s.t. } \forall\, \kappa > \kappa' : \Pr\left[\begin{matrix}\xi.\pi\,(T, 1^\kappa) = \perp\ : \\ T \leftarrow \mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(1^\kappa)\end{matrix}\right] < \xi.\beta(1^\kappa) + \kappa^c \tag{3}$$

2.4.2 AuthComRcv : authenticated-sender communication specification predicate

As an example of a specification predicate, we give the AuthComRcv predicate. For more specification predicates, see section 5.

The AuthComRcv model predicate verifies the authentic-sender property for all 'Incoming' events, which means that $out[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ indeed sent this message to $ent[\hat{e}_{\mathrm{R}}]$, during some previous event $\hat{e}_{\mathrm{S}} < \hat{e}_{\mathrm{R}}$. The AuthComRcv model predicate is shown in Algorithm 3.

Algorithm 3 AuthComRcv $(T,\ p)$ Predicate

```
1:  return (
2:    ∀ê_R  s.t.  1 ≤ ê_R ≤ T.e:
3:        if T.opr[ê_R] = 'Receive':                    ▷ For each 'Receive' event
4:            and T.out[ê_R] ≠ ⊥                        ▷ If the 'Receive' event was successful
5:            and T.ent[ê_R], T.out[ê_R].i_S ∈ T.N − T.F ▷ And both receiver and sender are honest
6:        then ∃ê_S < ê_R                               ▷ Then there is a previous event
7:            s.t.  T.opr[ê_S] = 'Send':                ▷ Which was a 'Send' event
8:            and T.inp[ê_S].m = T.out[ê_R].m           ▷ Where the input message was T.out[ê_R].m
9:            and T.inp[ê_S].i_R = T.ent[ê_R]           ▷ And where the intended recipient was T.ent[ê_R]
10:           and T.ent[ê_S] = T.out[ê_R].i_S           ▷ And where the entity was the sender output in event ê_R
      )
```

Usually, the base function used with the AuthComRcv model predicate would be $\beta = 2^{-l}$, where l is the length of some tags or signatures used by the protocol or scheme for authentication. In other words, we would usually allow the adversary to have probability 2^{-l} to cause an entity to receive a forged message, but not (significantly) more than 2^{-l}.

3 Confidentiality, Integrity, and Availability Models and Specifications

The execution process $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$ allows the adversary to choose all of the inputs to $\mathcal{P}$ during the execution process. However, definitions of properties such as confidentiality often include randomized inputs (for instance, to allow 'indistinguishability game'-based definitions). Thus, we define an extended execution process, $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p, b)$, which allows such indistinguishability-based definitions and specifications. Note that the only difference between $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$ and $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p, b)$ is the additional input bit and an additional special operation 'Challenge' which uses this bit. Consequently, we can use $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p, b)$ also to run executions equivalent to those of $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$ (i.e., simulate

$\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$) simply by using an adversary model in which $\mathcal{A}$ does not invoke the 'Challenge' operation. Therefore, $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p,b)$ is an extension of $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$.

3.1 The Exec_CIA Process

$\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p,b)$, shown in Algorithm 4, takes an additional input bit b that is not controlled or visible to $\mathcal{A}$. Secondly, as part of **Exec_CIA**, we define an additional special operation 'Challenge'. $\mathcal{A}$ can chose the 'Challenge' operation and specify two values in $inp[\hat{e}]$ (line 11 of Algorithm 4). The first value, $inp[\hat{e}][0]$, should be one of the operations of $\mathcal{P}$. The second input, $inp[\hat{e}][1]$, should be a tuple of two lists, where each list contains possible inputs to the operation specified in $inp[\hat{e}][0]$. (As usual, further assumptions about the inputs can be specified using a model.) Then $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p,b)$ invokes $\mathcal{P}$ with $inp[\hat{e}][0]$ as the operation, the b^{th} component of $inp[\hat{e}][1]$ (indexed from 0) as the input, and the state and clock values specified by $\mathcal{A}$ as before. Note that $\mathcal{A}$ does not know which element of $inp[\hat{e}][1]$ was sent to $\mathcal{P}$ as the input.

$\mathcal{A}$ can use the output of this operation, as well as the outputs of other operations executed during the process, to guess the value of b (0 or 1). (For example, $\mathcal{P}$ might output the encryption of $inp[\hat{e}][1][b]$, and $\mathcal{A}$ might try to guess b based on this encryption.) As before, $\mathcal{A}$ sets the value of $out_{\mathcal{A}}$ at the end of each event, and the final value of $out_{\mathcal{A}}$ is returned as part of the execution transcript T. Consequently, we can define specifications in terms of the probability that $T.out_{\mathcal{A}}$ equals b. (Usually, we do not want $\Pr[T.out_{\mathcal{A}} = b]$ to be significantly better than for a random guess of the value of b.)

3.2 Specifications

Specifications are the same as defined in subsection 2.2, except that the specification predicates take *three* inputs (including a bit b). We give an example of such a specification predicate in subsection 3.4.

3.3 Models

In definition 3, we define *adversary $\mathcal{A}$ CIA-satisfies model $\mathcal{M}$ with negligible advantage*. The differences between the asymptotic definitions of 'IA-satisfying' and 'CIA-satisfying' a model are that in the 'CIA-satisfying' definition, the model predicate takes the additional input bit b, and the probability is taken over a transcript returned by $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p,b)$ (instead of $\mathbf{Exec\_IA}_{\mathcal{A},\mathcal{P}}(p)$), for a random bit b.

Definition 3 (Adversary $\mathcal{A}$ CIA-satisfies model $\mathcal{M}$ with negligible advantage). *Let $\mathcal{A}$ be a PPT algorithm, and let $\mathcal{M} = (\mathcal{M}.\pi, \mathcal{M}.\beta)$ be a model, where $\mathcal{M}.\pi$ is a predicate over (T,p,b) and $\mathcal{M}.\beta$ is a function $\mathcal{M}.\beta(1^\kappa) : \{1\}^* \to [0,1]$. We say that adversary $\mathcal{A}$ CIA-satisfies model $\mathcal{M}$ with negligible advantage, denoted as $\mathcal{A} \models_{poly} \mathcal{M}$ (note that we reuse the IA notation for simplicity), if for*

Algorithm 4 $\textbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p, b)$
Adversary-Driven Execution Process with Challenge Bit

$(s_{\mathcal{A}}, \mathsf{N}) \leftarrow \mathcal{A}(p)$ ▷ *Initialize adversary*

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

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

repeat

  $e \leftarrow e + 1$ ▷ *Advance loop's counter*

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

  if $opr[e]$ ='Set-state' **then** ▷ *When $\mathcal{A}$ wants to change the local state of entity $ent[e]$*

    $s_{ent[e]}, out[e], sec\text{-}out[e][\cdot] \leftarrow inp[e], \perp, \perp$

  else if $opr[e]$ ='Set-output' **then** ▷ *When $\mathcal{A}$ wants to change the output of entity $ent[e]$*

    $out[e], sec\text{-}out[e][\cdot] \leftarrow inp[e]$

  else if $opr[e]$ ='Get-state' **then** ▷ *When $\mathcal{A}$ wants to get the current local state of entity $ent[e]$*

    $out[e], sec\text{-}out[e][\cdot] \leftarrow s_{ent[e]}, \perp$

  else if $opr[e]$ ='Sec-in' **then** ▷ *When $\mathcal{A}$ wants entity $ent[e]$ to receive secure output*
  ▷ *$\mathcal{A}$ specifies a previous event in $inp[e]$ and then the process uses $(ent[inp[e]], sec\text{-}out[inp[e]][ent[e]])$ as input to $\mathcal{P}$*

    $(s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}(s_{ent[e]}, \text{'Sec-in'}, (ent[inp[e]], sec\text{-}out[inp[e]][ent[e]]), clk[e])$

    ▷ *$\mathcal{P}$ returns the state of entity $ent[e]$, the output, and the secure output*

  else if $opr[e] =$ 'Challenge' **then** ▷ *When $\mathcal{A}$ wants a challenge, $\mathcal{A}$ specifies the operation in $inp[e][0]$ and two inputs in $inp[e][1]$, but only the b^{th} component of $inp[e][1]$ (indexed from 0) is used as input to $\mathcal{P}$*

    $(s_{ent[e]}, out[e], sec\text{-}out[e][\cdot]) \leftarrow \mathcal{P}(s_{ent[e]}, inp[e][0], inp[e][1][b], clk[e])$

  else ▷ *Otherwise, $\mathcal{A}$ wants to execute operation $opr[e]$ of $\mathcal{P}$ with input $inp[e]$ over entity $ent[e]$ with local clock of $clk[e]$*

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

  end if

  $(s_{\mathcal{A}}, out_{\mathcal{A}}, \mathsf{F}) \leftarrow \mathcal{A}(s_{\mathcal{A}}, out[e])$ ▷ *Inform $\mathcal{A}$ of the value of $out[e]$ and allow $\mathcal{A}$ to decide whether to continue ($out_{\mathcal{A}} = \perp$), or to terminate the loop ($out_{\mathcal{A}} \neq \perp$)*

until $out_{\mathcal{A}} \neq \perp$

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

Return T

every protocol $\mathcal{P}$, every constant c, and $b \xleftarrow{\$} \{0,1\}$, there is some $\kappa' \in \mathbb{N}$ such that for all $\kappa > \kappa'$, the probability that the model predicate $\mathcal{M}.\pi$, applied to a random resulting **Exec_CIA** *execution transcript T, security parameter 1^κ, and $b \in \{0,1\}$, is not satisfied, is less than $\mathcal{M}.\beta(1^\kappa) + \kappa^c$,*

$$\mathcal{A} \models_{poly} \mathcal{M} \stackrel{def}{=} \forall\ \mathcal{P}, c, b \xleftarrow{\$} \{0,1\}, \exists\ \kappa' \in \mathbb{N}\ s.t.\ \forall\ \kappa > \kappa' : \\ \Pr\left[\begin{matrix}\mathcal{M}.\pi\left(T, 1^\kappa, b\right) = \perp\ : \\ T \leftarrow \mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(1^\kappa, b)\end{matrix}\right] < \mathcal{M}.\beta(1^\kappa) + \kappa^c \tag{4}$$

3.4 Security Specifications

In definition 4, we define *protocol $\mathcal{P}$ CIA-satisfies specification ξ with negligible advantage under model $\mathcal{M}$*. Similarly to CIA models, the specification predicate takes the additional input bit b, and the probability is taken over a transcript returned by $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(p, b)$, for a random bit b.

Definition 4 (Protocol $\mathcal{P}$ CIA-satisfies specification ξ with negligible advantage under model $\mathcal{M}$). *Let $\xi = (\xi.\pi, \xi.\beta)$ be a specification, where $\xi.\pi$ is a predicate over (T, p, b) and $\xi.\beta$ is a function $\xi.\beta(1^\kappa) : \{1\}^* \to [0,1]$. We say that protocol $\mathcal{P}$ CIA-satisfies specification ξ with negligible advantage under model $\mathcal{M}$, denoted $\mathcal{P} \models^{\mathcal{M}}_{poly} \xi$ (note that we reuse the IA notation for simplicity), if for every PPT adversary $\mathcal{A}$ that CIA-satisfies $\mathcal{M}$ and $b \xleftarrow{\$} \{0,1\}$, for every constant c, there is some $\kappa' \in \mathbb{N}$ such that for all $\kappa > \kappa'$, the probability that the specification predicate $\xi.\pi$, applied to a random resulting* **Exec_CIA** *execution transcript T, security parameter 1^κ, and $b \in \{0,1\}$, is not satisfied, is less than $\xi.\beta(1^\kappa) + \kappa^c$, i.e.:*

$$\mathcal{P} \models^{\mathcal{M}}_{poly} \xi \stackrel{def}{=} \forall\ \mathcal{A} \in PPT, b \xleftarrow{\$} \{0,1\}, c,\ \textit{if}\ \mathcal{A} \models_{poly} \mathcal{M}\ \textit{then}\ \exists\ \kappa' \in \mathbb{N}\ s.t.\ \forall\ \kappa > \kappa' : \\ \Pr\left[\begin{matrix}\xi.\pi\left(T, 1^\kappa, b\right) = \perp\ : \\ T \leftarrow \mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(1^\kappa, b)\end{matrix}\right] < \xi.\beta(1^\kappa) + \kappa^c \tag{5}$$

As mentioned previously, we can now define specifications in terms of the probability that $T.out_{\mathcal{A}}$ equals b. In this way, we define the Indistinguishability Specification Predicate IND^π in Algorithm 5.

Algorithm 5 $\mathsf{IND}^\pi(T, p, b)$ Predicate

```
1: return ¬(
2:     T.out_A = b                    ▷ A guessed the value of b correctly
3:     and  π(T, p, b)
       )
```

By using specific predicates π, we can use the IND^π specification predicate to define more specific specification predicates; for example, we use the MsgConf

predicate, shown below, to define $\mathsf{IND}^{\mathsf{MsgConf}}$, which is used to define message confidentiality for the SecCom protocol.

The MsgConf Predicate

The MsgConf model predicate is given in Algorithm 6. The model 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'). We assume that $\mathcal{P}$ includes a 'Send' operation, which takes as input a message and the identifier of the recipient.
- During all 'Send' challenges, messages are only sent from one specific entity i_{S} to one specific entity i_{R}
- During each 'Send' challenge, $\mathcal{A}$ specifies two messages of equal length and the same recipient in the two possible inputs. That is, $\mathcal{A}$ specifies $inp[\hat{e}][1] = ((m_0, i_{\mathrm{R}}), (m_1, i_{\mathrm{R}}))$, where $|m_0| = |m_1|$. This ensures that $\mathcal{A}$ does not distinguish the messages based on their lengths.
- $\mathcal{A}$ does not use the 'Receive' operation to decrypt any output of a 'Send' challenge.

Algorithm 6 MsgConf (T, p, b) Predicate

```
1:  return (
2:    ∃ i_S, i_R s.t. :                                  ▷ There is one specific sender i_S and one specific receiver i_R
3:        ∀ê s.t. 1 ≤ ê ≤ T.e:
4:            if T.opr[ê] = 'Challenge'  then            ▷ S.t. in any 'Challenge' event
5:                T.inp[ê][0] = 'Send'                   ▷ It is a 'Send' challenge
6:                and T.ent[ê] = i_S                     ▷ The entity is i_S
7:                and |T.inp[ê][1][0].m| = |T.inp[ê][1][0].m|   ▷ A chose two messages of equal length
8:                and T.inp[ê][1][0].i_R = T.inp[ê][1][0].i_R = i_R   ▷ And the recipient for both messages is i_R
9:                and ∄ ê' s.t.  T.opr[ê'] = 'Receive'   ▷ And A did not use a 'Receive' event to decrypt the output of the challenge
10:                   and T.inp[ê'].c = T.out[ê].c
11:                   and T.ent[ê'] = i_R
12:                   and T.inp[ê'].i_S = i_S
      )
```

4 Concrete Security Models and Specifications

We extend the framework to allow concrete security definitions.

4.1 The Exec Process

The execution transcripts returned by **Exec_IA** and **Exec_CIA** already allow for specifications that refer to the concrete security parameters such as block

length, key length, and number of queries; however, they do not provide information about the time (number of steps) taken by the adversary when it is invoked during the execution. For this reason, we define an execution process **Exec**, which runs **Exec_CIA** but additionally ensures that the total number of steps taken by $\mathcal{A}$ is returned as part of the execution transcript.

To count the number of steps of $\mathcal{A}$, we use $\mathsf{StepCount}$, which runs $\mathcal{A}$ and counts the number of steps taken by $\mathcal{A}$. Namely:

$$\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(p,b) = \mathbf{Exec\_CIA}_{\mathsf{StepCount}(\mathcal{A}),\mathcal{P}}(1^\kappa, b) \tag{6}$$

Whenever **Exec_CIA** invokes the adversary with some input, e.g. $s_{\mathcal{A}}$, then $\mathsf{StepCount}(\mathcal{A})(s_{\mathcal{A}})$ is called. Then, $\mathsf{StepCount}$ gives $s_{\mathcal{A}}$ as input to $\mathcal{A}$, runs $\mathcal{A}$, and adds the number of steps taken by $\mathcal{A}$ to a counter that keeps track of the total number of steps taken by $\mathcal{A}$. If $\mathcal{A}$ outputs $\perp$, $\mathsf{StepCount}(\mathcal{A})$ also outputs $\perp$. When $\mathcal{A}$ finally outputs a value $\neq \perp$ to end the execution, then $\mathsf{StepCount}(\mathcal{A})$ returns this value *and* the total number of steps taken by $\mathcal{A}$. Consequently, the resulting execution transcript T is the same as if $\mathbf{Exec\_CIA}_{\mathcal{A},\mathcal{P}}(1^\kappa, b)$ had been run, except that $T.out_{\mathcal{A}}$ additionally contains the total number of steps taken by $\mathcal{A}$.

4.2 Specifications

Specifications are defined as before, but now we finally use the advantage function as well, and we no longer assume that the parameter p is a unary security parameter. Instead, depending on the particular protocol or scheme under consideration, p may contain, e.g., numbers of queries, block length, and key length. Notice that this allows specifications to include restrictions on various parameters, including numbers of queries and the number of steps that $\mathcal{A}$ takes. We use *Params* to indicate the set of possible (allowed) values for p. This is also the domain of both the base function and the advantage function.

We say that (a protocol or an adversary) *satisfies ξ with advantage ϵ* if, for a random resulting execution transcript T, $b \xleftarrow{\$} \{0,1\}$, and $p \in Params$, the probability that $\xi.\pi(T,p,b)$ returns $\perp$ is bounded by $\xi.\beta(p) + \epsilon(p)$.

4.3 Models

Definition 5 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with advantage ϵ). *Let $\mathcal{A}$ be a PPT algorithm, and let $\mathcal{M} = (\mathcal{M}.\pi, \mathcal{M}.\beta)$ be a model, where $\mathcal{M}.\pi$ is a predicate (i.e., outputs $\top$ or $\perp$) and $\mathcal{M}.\beta$ is a function $\mathcal{M}.\beta(p) : Params \to [0,1]$, where Params is the set of possible values of the parameters. Furthermore, let ϵ also be a function $\epsilon(p) : Params \to [0,1]$, such that, for any given p, holds $\mathcal{M}.\beta(p) + \epsilon(p) \in [0,1]$. We say that adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ with advantage ϵ, denoted as $\mathcal{A} \models_\epsilon \mathcal{M}$, if for every protocol $\mathcal{P}$, every $p \in Params$, and $b \xleftarrow{\$} \{0,1\}$ the probability that the model predicate $\mathcal{M}.\pi$, applied to a random*

resulting execution transcript T, $p \in Params$, and $b \xleftarrow{\$} \{0,1\}$ is not satisfied, is bounded by $\xi.\beta(p) + \epsilon(p)$, i.e.:

$$\mathcal{A} \models_{\epsilon} \mathcal{M} \stackrel{def}{=} \forall\ \mathcal{P}, p \in Params, b \xleftarrow{\$} \{0,1\} : \\ \Pr\left[\begin{matrix}\mathcal{M}.\pi\,(T,p,b) = \perp\ : \\ T \leftarrow \mathbf{Exec}_{\mathcal{A},\mathcal{P}}(p,b)\end{matrix}\right] \leq \mathcal{M}.\beta(p) + \epsilon(p) \tag{7}$$

4.4 Security Specifications

Definition 6 (Protocol $\mathcal{P}$ satisfies specification ξ with advantage ϵ under model $\mathcal{M}$). *Let $\xi = (\xi.\pi, \xi.\beta)$ be a specification, where $\xi.\pi$ is a predicate over (T,p,b) and $\xi.\beta$ is a function $\xi.\beta(p) : Params \to [0,1]$, where $Params$ is the set of possible values of the parameters. Furthermore, let $\xi.\epsilon$ also be a function $\xi.\epsilon(p) : Params \to [0,1]$, such that, for any given p, holds $\xi.\beta(p) + \epsilon(p) \in [0,1]$, and let $\mathcal{M}$ be a model. We say that protocol $\mathcal{P}$ satisfies specification ξ with advantage ϵ under model $\mathcal{M}$, denoted $\mathcal{P} \models_{\epsilon}^{\mathcal{M}} \xi$, if for every PPT adversary $\mathcal{A}$ that CIA-satisfies $\mathcal{M}$, every $p \in Params$, and $b \xleftarrow{\$} \{0,1\}$, the probability that the specification predicate $\xi.\pi$, applied to a random resulting execution transcript T, $p \in Params$, and $b \in \{0,1\}$, is not satisfied, is bounded by $\xi.\beta(p) + \epsilon(p)$, i.e.:*

$$\mathcal{P} \models_{\epsilon}^{\mathcal{M}} \xi \stackrel{def}{=} \forall\ \mathcal{A} \in PPT, p \in Params, b \xleftarrow{\$} \{0,1\},\ \textit{if } \mathcal{A} \models \mathcal{M} \textit{ then}: \\ \Pr\left[\begin{matrix}\xi.\pi\,(T,p,b) = \perp\ : \\ T \leftarrow \mathbf{Exec}_{\mathcal{A},\mathcal{P}}(p,b)\end{matrix}\right] \leq \xi.\beta(p) + \epsilon(p) \tag{8}$$

5 Further Specification Predicates

5.1 Model Predicates

5.1.1 $\mathcal{M}_{\mathbf{SecInit}}^{\hat{r}\text{-}\mathbf{rounds}}$: the $\hat{r}$-rounds secure initialization model predicate

Cryptographic protocols are often designed assuming a secure initialization process, e.g., assuming shared secret keys. However, in the execution process (Algorithm 1), entities can only communicate via the adversary. As a result, we cannot simply *assume* shared secret keys, but the entities can use their local randomness to generate secret keys, and they can communicate, using cryptography, to securely establish shared secret values. We next define a simple secure initialization model predicate, $\mathcal{M}_{\text{SecInit}}^{\hat{r}\text{-rounds}}$. This model predicate ensures $\hat{r}$ secure 'rounds' of $|\mathsf{N}|$ events each, where in event $\hat{e}$ (where $1 \leq \hat{e} \leq \hat{r} \cdot |\mathsf{N}|$) holds:

- Entities are invoked with the special operation 'Init', i.e., $opr[\hat{e}]$ = 'Init', and in 'round robin', i.e., $ent[\hat{e}] = \hat{e} \mod |\mathsf{N}|$ (where $\mathsf{N} = \{1,2,\ldots\}$). Note, in particular, that this prevents the adversary, during the initialization, from invoking the special 'Set-state', 'Set-output', and 'Get-state' operations to control the state or output of an entity ('Set-state' or 'Set-output') or to expose the state of an entity ('Get-state').

– Authenticated, reliable communication. Namely, every message received by entity i_{R} from entity i_{S} at round $2 \leq r \leq \hat{r}$, was indeed sent by i_{S} in the previous round to i_{R}; and vice verse, i.e., every message sent by i_{S} to i_{R} at round $1 \leq r \leq (\hat{r}-1)$, is correctly received by i_{R}, from sender i_{S}, in the next round.

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

$$\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{SecInit}}(T, 1^\kappa) = \mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitOps}}(T, 1^\kappa) \wedge \mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitCom}}(T, 1^\kappa) \tag{9}$$

where $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitOps}}$ captures the first aspect ('operations') and $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ captures the second aspect ('communications'). We now define each of these more precisely; for convenience, let $\mathsf{N} = \{1, 2, \ldots\}$. The $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitOps}}$ predicate is shown in Algorithm 7.

Algorithm 7 $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitOps}}(T, p)$ Predicate

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

To define the $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ (sub)model predicate, we assume the following conventions. To cause an entity to receive a message, the adversary includes the triple $(\text{'auth\_init\_recv'}, m, i_{\mathrm{S}})$ (where m is the message and $i_{\mathrm{S}} \in \mathsf{N}$ is the purported sender) as part of the input $inp[\hat{e}_{\mathrm{R}}]$. The sender indicates the sending of message m to $i_{\mathrm{R}} \in \mathsf{N}$ by specifying, as part of its output $out[\hat{e}_{\mathrm{S}}]$, a triplet $(\text{'auth\_init\_send'}, m, i_{\mathrm{R}})$. The $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitCom}}$ predicate is shown in Algorithm 8.

Algorithm 8 $\mathcal{M}^{\hat{r}\text{-rounds}}_{\text{InitCom}}(T, p)$ Predicate

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

5.1.2 $\mathcal{M}^{|\mathsf{F}| \leq f}$: up to f Byzantine (malicious) faults model predicate

We next define $\mathcal{M}^{|\mathsf{F}| \leq f}$, a specific adversary model predicate allowing the adversary to choose, and completely control, up to f of the entities in N. We refer

to such failures, where the adversary is allowed to completely control the entity, as *malicious* or *Byzantine* faults. We use f as a function applied to the total number of entities $|\mathsf{N}|$. We refer to this particular faults model as $\mathcal{M}^{|\mathsf{F}|\leq f}$, where $f : \mathbb{N} \to \mathbb{N}$ bounds the number of faulty entities as a function of the total number of entities. Specifically, the adversary may corrupt entities by performing the 'Get-state', 'Set-state', and 'Set-output' operations. To enforce the model predicate, we simply ensure that the 'Get-state', 'Set-state', and 'Set-output' operations can be applied only to entities in F, and that $|\mathsf{F}| \leq f(|\mathsf{N}|)$. We define the $\mathcal{M}^{|\mathsf{F}|\leq f}$ predicate in Algorithm 9.

Algorithm 9 $\mathcal{M}^{|\mathsf{F}|\leq f}$ $(T,\, p)$ Predicate

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

PoC-PKI, our proof-of-concept PKI, uses $\mathcal{M}^{|\mathsf{F}|\leq f}$ with f defined as: $f(n) = \lfloor (n/3) \rfloor$. Namely, the faulty entities can arbitrarily misbehave so long the adversary controls at most a third of all entities. Note that $\mathcal{M}^{|\mathsf{F}|\leq f}$ only restricts the set F of faulty entities, which does not yet restrict the adversary's ability to interfere with the communication, clock, and local inputs of any entity - including non-faulty entities (i.e., in $\mathsf{N} - \mathsf{F}$). By using $\mathcal{M}^{|\mathsf{F}|\leq f}$ in conjunction with additional model predicates, e.g., $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$ and $\mathcal{M}^{\text{CLK}}_{\Delta_{clk}}$ (described next), we can also restrict the adversary in such additional ways, as desired for a particular analysis. Notice, in particular, that $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$, which defines restrictions on the communication events, completely excludes events where the sender or recipient are faulty (i.e., in F). This allows the adversary to completely control all of faulty entities.

5.1.3 $\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}$: authentic-sender, bounded-delay communication model predicate

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

$$\mathcal{M}^{\text{AuthCom}}_{\Delta_{com}}(T, 1^\kappa) = \mathcal{M}^{\text{Com}}_{\Delta_{com}}(T, 1^\kappa) \wedge \mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}}(T, 1^\kappa) \tag{10}$$

We first present $\mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}}$, which ensures authentic-sender for message-receive events. The adversary decides on the function $opr[\hat{e}_{\text{R}}]$ to be invoked at

every event $\hat{e}_{\mathrm{R}}$ as well as the input $inp[\hat{e}_{\mathrm{R}}]$. We assume a convention for *send* and *receive* events as follows. The adversary causes a message receipt event by setting $opr[\hat{e}_{\mathrm{R}}]$ to 'Receive' and $inp[\hat{e}_{\mathrm{R}}]$ to (m, i_{S}) (where m is the message and $i_{\mathrm{S}} \in \mathsf{N}$ is the purported sender). We use dot notation to refer to the message ($inp[\hat{e}_{\mathrm{R}}].m$) and to the sender ($inp[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$). Also, we allow the sender $ent[\hat{e}_{\mathrm{S}}]$ to specify, as part of its output $out[\hat{e}_{\mathrm{S}}]$, one or more triplets of the form $(\text{'send'}, m, i_{\mathrm{R}})$, indicating the sending of message m to $i_{\mathrm{R}} \in \mathsf{N}$.

The authentic-sender property ($\mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate) implies that $inp[\hat{e}_{\mathrm{R}}].i_{\mathrm{S}}$ indeed sent this message to $ent[\hat{e}_{\mathrm{R}}]$, during some previous event $\hat{e}_{\mathrm{S}} < \hat{e}_{\mathrm{R}}$. The $\mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}}$ model predicate is shown in Algorithm 10.

Algorithm 10 $\mathcal{M}^{\text{AuthCom-rcv}}_{\Delta_{com}}$ $(T,\, p)$ Predicate

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

The $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate ensures reliable, bounded-delay delivery of messages sent. Assume that at event $\hat{e}_{\mathrm{S}}$ of the execution, the output $out[\hat{e}_{\mathrm{S}}]$ generated by $ent[\hat{e}_{\mathrm{S}}]$, includes a $(\text{'send'}, m, j)$ triplet, i.e., $ent[\hat{e}_{\mathrm{S}}]$ sends message m to $j \in \mathsf{N}$. If the $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate is true for this execution, then after at most Δ_{com}, if the execution did not terminate already, then entity j would receive m from $ent[\hat{e}_{\mathrm{S}}]$. The $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ model predicate is shown in Algorithm 11.

Algorithm 11 $\mathcal{M}^{\text{Com}}_{\Delta_{com}}$ $(T,\, p)$ Predicate

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

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

5.1.4 $\mathcal{M}^{\text{CLK}}_{\Delta_{clk}}$: bounded-drift clock synchronization assumptions

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

$$\mathcal{M}^{\text{CLK}}_{\Delta_{clk}}(T,1^\kappa) = \mathcal{M}^{\text{Drift}}_{\Delta_{clk}}(T,1^\kappa) \wedge \mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}(T,1^\kappa) \tag{11}$$

We begin with $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$, which bounds the clock drift. It enforces two requirements on the execution: each local-clock value ($clk[\hat{e}]$) must be within Δ_{clk} drift from the real time $\tau[\hat{e}]$, and the real time values should be monotonously increasing. See Algorithm 12.

Algorithm 12 $\mathcal{M}^{\text{Drift}}_{\Delta_{clk}}$ $(T,\ p)$ Predicate

```
1: return (
2:     ∀ê s.t 1 ≤ ê ≤ 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
5:             if ê ≥ 2                             ▷ And if ê is not the first event
6:             then T.τ[ê] ≥ T.τ[ê − 1]              ▷ Then the real time is ≥ the real time at the previous event
    )
```

As a special case, when $\Delta_{clk} = 0$, this function defines a model where the local clocks are fully synchronized, i.e., there is no difference between entities' clocks. Finally, $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$ provides a 'wake-up service' allowing the protocol to perform time-driven activities and ensuring that appropriate functions are invoked properly. This is ensured by requiring that if $(\textit{'Sleep'}, x)$ was part of the output $out[\hat{e}]$ (indicating that entity $ent[\hat{e}]$ was 'put to sleep' for x time) and execution did not terminate by 'real' time $\tau[\hat{e}] + x + \Delta_{clk}$, then at some event $\hat{e}' > \hat{e}$ (where $\tau[\hat{e}']$ was within Δ_{clk} from $\tau[\hat{e}] + x$), the same entity ($ent[\hat{e}]$) was indeed 'Woken up'. The $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$ predicate appears in Algorithm 13.

Algorithm 13 $\mathcal{M}^{\text{Wake-up}}_{\Delta_{clk}}$ $(T,\, p)$ Predicate

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

5.1.5 Securely shared keys ($\mathcal{M}_{\text{SecKeyShare}}$) specification predicate

Sometimes, especially when analyzing shared-key protocols, it may be useful to *assume* that the entities securely share keys before communicating with each other - i.e., to assume that the keys are correctly shared without the interference of the adversary. For this purpose, we define the model $\mathcal{M}_{\text{SecKeyShare}}$, shown in Algorithm 14, which ensures that values (keys) are securely shared (using the *sec-out* mechanism in the execution process) before entities send or receive messages. Specifically, before any 'Send' or 'Receive' event, the sender of the message must have securely shared values with the receiver, and the receiver of the message must have securely shared values with the sender. We assume that the operation 'Sec-channel-setup' is used to share these values.

5.1.6 Verifiable Attribution of Statements (VAS) specification predicate

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

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

Algorithm 14 $\mathcal{M}_{\text{SecKeyShare}}(T,\, p)$ Predicate

return $\top$ if (

$\forall \hat{e}''$ **s.t.** $T.opr[\hat{e}'']$ = 'Send' **or** $T.opr[\hat{e}'']$ = 'Receive': ▷ *For each message 'Send' event and each message 'Receive' event*

  if $T.opr[\hat{e}'']$ = 'Send' ▷ *Determine the sender and receiver of the message*

  then $i_{\text{S}}, i_{\text{R}} \leftarrow T.ent[\hat{e}''], T.inp[\hat{e}''].i_{\text{R}}$

  else if $T.opr[\hat{e}'']$ = 'Receive'

  then $i_{\text{S}}, i_{\text{R}} \leftarrow T.inp[\hat{e}''].i_{\text{S}}, T.ent[\hat{e}'']$

  $\exists\, \hat{e}, \hat{e}'$

    s.t. $\hat{e} < \hat{e}' < \hat{e}''$ ▷ *Previously (before the 'Send' or 'Receive' event)*

    and $T.opr[\hat{e}]$ = 'Sec-channel-setup' ▷ *Keys were securely sent*

    and $T.ent[\hat{e}] = i_{\text{S}}$ ▷ *From the sender of the message*

    and $T.sec\text{-}out[\hat{e}][i_{\text{R}}] \neq \perp$ ▷ *To the receiver of the message*

    and $T.opr[\hat{e}']$ = 'Sec-in' ▷ *And they were securely received*

    and $T.ent[\hat{e}'] = i_{\text{R}}$ ▷ *By the receiver of the message*

    and $T.inp[\hat{e}'] = \hat{e}$

  and $\exists\, \hat{e}, \hat{e}'$

    s.t. $\hat{e} < \hat{e}' < \hat{e}''$ ▷ *And also previously (before the 'Send' or 'Receive' event)*

    and $T.opr[\hat{e}]$ = 'Sec-channel-setup' ▷ *Keys were securely sent*

    and $T.ent[\hat{e}] = i_{\text{R}}$ ▷ *From the receiver of the message*

    and $T.sec\text{-}out[\hat{e}][i_{\text{S}}] \neq \perp$ ▷ *To the sender of the message*

    and $T.opr[\hat{e}']$ = 'Sec-in' ▷ *And securely received*

    and $T.ent[\hat{e}'] = i_{\text{S}}$ ▷ *By the sender of the message*

    and $T.inp[\hat{e}'] = \hat{e}$

)

We formalize this by assuming that each entity $i \in \mathsf{N}$ *identifies* its public key pk by outputting the pair $(\text{'public key'}, pk) \in out[\hat{e}]$, in some event $\hat{e}$; namely, we use 'public key' as a 'label', to identify output of the public key. Typically, entities output the public key when they generate the key, i.e., $ent[\hat{e}] = i$, possibly as an initialization operation, i.e. $opr[\hat{e}] = \textit{'Init'}$. Notice that entities may often also send their public keys to each other using the $(\text{'send'}, m, i_{\mathrm{R}})$ output convention described in § 5.1.3; however, we prefer to keep the two conventions separate, since we believe that not every protocol that uses verification of attribution would necessarily send public keys in precisely the same way.

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

$$V_{ka}(i, pk, T) = \{\exists \hat{e} \ s.t. \ T.ent[\hat{e}] = i \ \wedge (\text{'public key'}, pk) \in T.out[\hat{e}]\} \tag{12}$$

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

The specification predicate is defined with respect to specific signature scheme $\mathcal{S}$, and the V_{ka} predicate defined above (Eq. 12). For simplicity, and since $\mathcal{S}$ is typically obvious (as part of $\mathcal{P}$), we do not explicitly specify $\mathcal{S}$ as a parameter of the specification predicate. The VAS predicate is shown in Algorithm 15.

Algorithm 15 Verifiable Attribution of Statements Predicate $\mathsf{VAS}(T, p)$

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

5.1.7 Generic misbehavior detection specification predicates

Many security protocols are required to be *resilient to misbehaviors*, i.e., to achieve their goals even if some of the entities, say entities in $\mathsf{F} \subset \mathsf{N}$, are faulty, and may misbehave (arbitrarily or in some specified manner). This resiliency to faulty, misbehaving entities is often based on *detection of misbehavior*; furthermore, often, many security protocols are required only to *detect misbehaviors*, which would be followed by taking some additional measures to deter and/or neutralize an attack. While misbehavior can be detected in different ways, detection is typically based either on some *evidence* that a certain entity is dishonest, where the evidence should be *verifiable by any third party*, or based on an *accusation*, where one entity (the *accuser*) accuses another entity (the *suspect*) of some misbehavior. Such an accusation may not be true, and therefore, it is harder to use this approach to deter and/or neutralize the attack; however, many misbehaviors do not leave any *evidence* verifiable by a third party, in which case, accusations may provide some security benefits, e.g., *detection* of the attack. A typical example of such misbehavior that does not leave any evidence is when a party *fails to act* in a required way, e.g., to send a required message or response; such failure may be plausibly blamed on communication issues, or on failure of the intended recipient. Often, a party, say Alice, detects such failure, say of Mal, to send a required message, after Alice waits for some *maximum delay*, and then Alice issues an 'accusation' against Mal, to alert others; for example, see [4]. An honest entity would only accuse a misbehaving party; however, because an accusation cannot be verified, a misbehaving entity could falsely accuse anyone, even an honest entity.

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

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

Let $V_{PoM} : \{0,1\}^* \times \{0,1\}^* \to \{\top, \bot\}$ be a predicate. The Non-frameability predicate, shown in Algorithm 16, returns $\bot$ if the adversary was able to output a Proof of Misbehavior for an honest entity, and $\top$ otherwise.

Algorithm 16 Non-frameability Predicate $\mathsf{NF}(T, p)$

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

5.1.8 Accusations and the No False Accusations specification predicate

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

Intuitively, the *No False Accusations (NFA)* specification predicate states that a non-faulty entity $a \notin \mathsf{F}$ (Alice), would *never (falsely) accuse* of a fault, another non-faulty entity, $b \notin \mathsf{F}$ (Bob). To properly define this specification predicate, we first define a convention for one party, say $a \in \mathsf{N}$ (for 'Alice'), to output an Indicator of Accusation, i.e., 'accuse' another party, say $i_{\mathrm{M}} \in \mathsf{N}$ (for 'Mal'), of a fault. Specifically, we say that at event $\hat{e}_A$ of the the execution, entity $ent[\hat{e}_A]$ *accuses* entity i_{M} (Mal), if $out[\hat{e}_A]$ is a triplet of the form $(\mathrm{IA}, i_{\mathrm{M}}, x)$. The last value in this triplet, x, should contain the clock value at the *first* time that Alice accused Mal; we discuss this in section 5 as the value x is not relevant for the specification predicate, and is just used as a convenient convention for some protocols.

The No False Accusations (NFA) predicate 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, $\mathsf{NFA}(T, 1^\kappa)$ 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., $j, T.ent[e] \in T.\mathsf{N} - T.\mathsf{F}$). See Algorithm 17.

Algorithm 17 No False Accusations Predicate $\mathsf{NFA}(T, p)$

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

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

To simplify the predicate, let $fc(i, i_{\text{M}}, T)$ be the value of $T.clk[\hat{e}]$, where $\hat{e}$ is the first event in T in which entity i accused entity $i_{\text{M}} \in T.\mathsf{N}$ (or $\perp$ if no such event exists). The Use First-Accuse Time (UFAT) predicate UFAT is defined in Algorithm 18.

Algorithm 18 Use First-Accuse Time Predicate $\mathsf{UFAT}(T, p)$

```
1: return ¬(
2:     T.ent[T.e] ∈ T.N − T.F                          ▷ T.ent[T.e] is an honest entity
3:     and ∃i_M ∈ T.N  s.t. (IA, i_M, x) ∈ T.out[T.e]
                       and x ≠ fc(T.ent[T.e], i_M, T)   ▷ T.ent[T.e] did not indicate the first time of accusation in an accusation
       )
```

6 Conclusions and Future Work

In this work, we discussed the lack of appropriate framework that allows rigorous analysis of applied, stateful cryptographic protocols, with different communication, synchronization and adversary models. We addressed this challenge by presenting the *Modular Specifications Security framework*, where security is defined with respect to a specific *model predicate* $\mathcal{M}$. The framework appears useful for various tasks; it allows comparison of protocols based on the specifications they satisfy and the models they assume. Definitions of models and specifications may be reused across different types of protocols and schemes; we identified several 'generic' specifications, which appear to be applicable to many different tasks.

We see this work as only the stepping stone to further research into the Modular Specifications Security framework. Specifically, into applying the framework onto various schemes, e.g., PKI schemes, in a way that will meet all desired specifications, minimizing or avoiding simplifications.

Another important challenge is to extend the Modular Specifications Security framework to support secure compositions, possibly following UC [1]. Specifically, it would be interesting to extend [2], which presents a UC definition for a simplified basic PKI (a reduced functionality of the X.509 PKI).

References

1. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. pp. 136–145. IEEE (2001), online at https://eprint.iacr.org/2000/067.pdf, last updated Dec. 2018.
2. 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
3. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The theory of timed i/o automata. Synthesis Lectures on Distributed Computing Theory **1**(1), 1–137 (2010)
4. Leibowitz, H., Piotrowska, A.M., Danezis, G., Herzberg, A.: No Right to Remain Silent: Isolating Malicious Mixes. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1841–1858 (2019)
5. Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. pp. 184–200. IEEE (2001)