CMoSS: Composable Modular Security Specifications Framework

Sara Wrótniak
School of Computing
University of Connecticut
Storrs, CT, USA
sara.wrotniak@uconn.edu

Hemi Leibowitz
Faculty of Computer Science
The College of Management Academic Studies
Rishon LeZion, Israel
menahemle@colman.ac.il

Ewa Syta
Department of Computer Science
Trinity College
Hartford, CT, USA
ewa.syta@trincoll.edu

Amir Herzberg
School of Computing
University of Connecticut
Storrs, CT, USA
amir.herzberg@uconn.edu

***Abstract*—CMoSS facilitates modular specifications, design and analysis of cryptographic protocols. Modular design and analysis is achieved by supporting provably-secure compositions of protocols; typically, a protocol uses a *blackbox subprotocol*, and is proven secure when composed with any subprotocol meeting the blackbox specifications. For modularity of specifications, CMoSS extends the approach of the MoSS framework: protocol specifications are defined modularly, by a set of independent predicates (games) for each *model* (assumption) and *requirement* (goal). CMoSS makes it feasible to rigorously specify, develop and analyze realistic applied cryptographic protocols, supporting real-time concurrency and involving different attacker capabilities, delays, faults and synchronization challenges. CMoSS specifications provide a precise formalization of the informal specifications used by practitioners, facilitating provable security for practical protocols.**

1. Introduction

Cryptographic protocols are critical to the security of systems and networks. To ensure security, we must clearly define both the protocol's goals and its intended operating environment. Often, the specifications are presented as informal, textual lists of requirements (goals) and models (assumptions about the operating environment).

This approach is *modular*, since each requirement and model is described separately. This results in simpler models and requirements, and allows the use of partially overlapping sets of models and requirements in different works, which is convenient, e.g., allows reuse and comparison.

However, an informal, textual list of requirements and models cannot be rigorously analyzed, and rigorous analysis is required to ensure security of complex systems and protocols.

Modern cryptography established, e.g., in [15], the gold standard: rigorous specifications and rigorous analysis (proof of security), often by reduction to an assumed-secure 'blackbox'. Many of these works use *game-based specifications*, where each security goal is defined as a game between an adversary and the protocol. The fact that each goal is (typically) mapped to a different game provides some modularity; however, this is still much less modular then the informal lists of textual requirements and assumptions/models.

The recently proposed MoSS framework (Modular Security Specifications) [17] is the first to facilitate rigorous analysis while preserving fully-modular specifications. Namely, in MoSS, protocols are proven to satisfy a set of requirements (goals), assuming a set of models (assumptions). Each model (assumption) and each requirement (goal) is defined rigorously and separately from the other requirements/models.

However, MoSS does not facilitate *modular design*, that is, *secure composition of provably secure protocols*. In fact, secure compositions are not supported by most game-based approaches. Exceptions include state separation proofs [4] and works that study secure compositions of specific protocols, such as parallel compositions of zero-knowledge protocols [13] and the composition of key exchange and authenticated encryption protocols [2], [3].

Additional frameworks which facilitate modular design of provably-secure cryptographic protocols use the approach of *simulation-based* security specifications. The most well known and widely used are Canetti's *Universal Composability (UC)* framework [6], [7] and its extensions, including iUC, GNUC, IITM, and simplified-UC [5], [18], [22], [26]; secure reactive systems [1]; and constructive cryptography [25], [24], [23]. However, in these frameworks, the *specifications* are inherently *non-modular*. For example, in UC and other simulation-based frameworks, a protocol must be indistinguishable from an ideal functionality that encapsulates all requirements and assumptions into a single, monolithic specification. As a result, even reasoning about

an ideal functionality, for example with just one requirement removed, becomes challenging.

In summary, existing frameworks support *either modular, game-based specifications* (as in MoSS) *or modular design and analysis* (as in UC), *but not both*.

We present *CMoSS (Composable Modular Security Specifications)*, a new framework for provable security that supports *both modular game-based specifications* (extending MoSS) and *modular design* (providing *composability*).

CMoSS further supports *realistic, powerful modeling of time, communication, failures and concurrency*. Supporting concurrency and realistic modeling of time is challenging, and combining these capabilities with provably secure compositions presents an even greater challenge. Most frameworks only support simplified models, mainly asynchronous; only few support synchronous clocks, see [23], [9]. Among existing frameworks for provably secure composition of protocols, only the secure reactive systems [1] framework allows realistic modeling of concurrency and time. In contrast, CMoSS supports *concurrency and realistic modeling of time* as well as *modular specifications and secure compositions*, while relying on game-based design, making it easier to understand and use.

Table 1 compares CMoSS with existing approaches to security specification, design, and analysis, and support for real-time concurrency.

TABLE 1. APPROACHES TO SPECIFICATION, DESIGN AND ANALYSIS OF SECURITY.

Approach	Specifications			Compositions (modular design)	Real time concurrency	Provable security
	Intuitive specs	Defined for each…	Modular specs			
Informal	✓	Goal & model	✓	✓	✓	✗
Game-based	✓	Goal	✗	✗	✗	✓
UC, iUC, …	✗	Scheme/protocol	✗	✓	✗	✓
Reactive systems	✗	Scheme/protocol	✗	✓	✓	✓
MoSS	✓	Goal & model	✓	✗	✗	✓
CMoSS	✓	Goal & model	✓	✓	✓	✓

Real-world Applications.. CMoSS is already being used by researchers to specify and analyze the security of important real-world protocols: RPKI (a manuscript of this work is currently in submission), TLS 1.3, and blockchains.[1]

Running Example.. Throughout the paper, we use clock synchronization as an example, to demonstrate how to apply CMoSS for a cryptographic protocol with non-trivial distributed-computing aspects[2]. Specifically, we show how to design a clock synchronization protocol $\mathcal{P}_{\text{UnauthCS}}$, by assuming an authenticated communication channel that $\mathcal{P}_{\text{UnauthCS}}$ would operate on, using a *black-box composition requirement* $\mathcal{R}_{\text{ARR}}$. We also describe a request-response protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$. Protocols $\mathcal{P}_{\text{UnauthCS}}$ and $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ can be composed into one protocol. See Figure 1.

1. The RPKI and TLS manuscripts can be made available to the reviewers .

2. We intentionally picked a protocol which uses simple cryptography, and the distributed-computing aspects are not very complex; more realistic protocols will deserve their own paper.

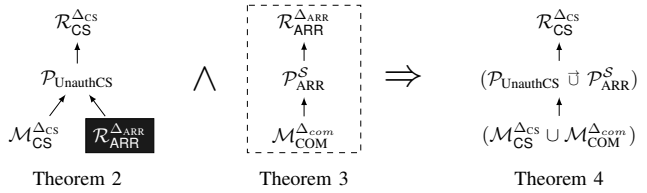

Figure 1. Illustration of the modular design of the clock synchronization example protocol. Theorem 2 states that protocol $\mathcal{P}_{\text{UnauthCS}}$ satisfies the clock synchronization requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$, assuming a model $\mathcal{M}^{\Delta_{\text{CS}}}_{\text{CS}}$ and a 'black-box' protocol satisfying the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$. Theorem 3 states that protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ satisfies the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$ assuming a bounded-delay communication model $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$. From these results and using Lemma 6, Theorem 4 states that when $\mathcal{P}_{\text{UnauthCS}}$ is composed with $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$, the composite protocol, denoted as $(\mathcal{P}_{\text{UnauthCS}} \,\tilde{\cup}\, \mathcal{P}^{\mathcal{S}}_{\text{ARR}})$, satisfies $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$ assuming both models, i.e., $(\mathcal{M}^{\Delta_{\text{CS}}}_{\text{CS}} \cup \mathcal{M}^{\Delta_{com}}_{\text{COM}})$, without assuming a black-box requirement.

Contributions.. CMoSS bridges the gap between provably secure cryptography and applied security engineering. Our key contributions are:

1) We present CMoSS, the first game-based framework that facilitates provable security and supports both *modular design* of protocols by composition as well as *modular and intuitive specifications*. CMoSS specifications are defined by models and requirements, each of which is defined by one or more rigorous yet intuitive predicates.
2) CMoSS is the first framework to support *real-time concurrency* for modular specifications, allowing concurrent execution of multiple protocols and multiple instances over a set of entities.
3) We prove that the composition of protocols is *associative*, and that the composition of *non-conflicting* protocols is *commutative*. Two protocols are conflicting if they have a subprotocol in common, but the subprotocol is defined differently in the two protocols.
4) We present *black-box compositions*, which allow modular design and analysis of protocols. Essentially, this involves designing a 'higher layer' protocol and proving its security when composed with any protocol satisfying a *black-box requirement*, and separately designing a 'lower layer' protocol satisfying the black-box requirement. We demonstrate the use of black-box composition by designing and analyzing a simple secure clock synchronization protocol, which assumes a black-box requirement for a request-response protocol with server authentication; separately, we design and analyze a simple request-response protocol which satisfies the blackbox requirement. This example is illustrated in Figure 1.

Organization.. §2 presents the CMoSS framework and its execution model. §3 defines how protocols are composed in CMoSS and presents composition properties. §4 presents black-box requirements, black-box composition lemmas, and the ask convention and model for interaction between subprotocols. §5 gives an overview of the running clock-synchronization example, which demonstrates

how CMoSS can be used for modular protocol design and analysis, and §6 concludes. The appendices include details and proofs.

2. The Composable Modular Security Specifications (CMoSS) Framework

In this section, we present CMoSS. Our description is self-contained. CMoSS builds on MoSS [17] in support of modular specifications, however, with significant extensions.

2.1. CMoSS Fundamentals

CMoSS facilitates game-based analysis of cryptographic protocols, with a key change: it separates the execution process from the specifications being analyzed. In a 'classical' game-based analysis, we define processes, called *games*, for the interactions between the protocol and the adversary. Each game defines the possible executions resulting from interactions between the adversary and the protocol. The game defines the interactions and controls available to the adversary, limiting the adversary's actions, information and control to enforce the models/assumptions made, e.g., preventing an eavesdropping adversary from modifying messages. A separate game is typically defined for each requirement, and the game evaluates and outputs a bit indicating whether the execution satisfied the requirement.

CMoSS also analyzes the results of interactions between the adversary and the protocol; however, for all protocols, requirements and assumptions (models), CMoSS uses *the same execution process ('game')*, denoted as $\mathbf{Exec}$ and described below. In CMoSS, the game does not enforce models or evaluate whether a requirement is satisfied. Instead, each requirement and model is defined using a *predicate*, whose input is the transcript of an execution of the protocol and the adversary. This yields *modular specifications*: each requirement and each model is defined by a separate predicate, without dependency on other requirements or models. This contrasts with 'classical' game based analysis, where each requirement is defined by a dedicated game that enforces a specific set of assumptions (models).

CMoSS's execution process is *adversary-driven*, meaning that the adversary determines what happens during an execution of a protocol, from selecting which operation to execute on which entities and at what time, to specifying all inputs, including local clock values, and accessing all outputs. However, when we analyze whether a protocol $\mathcal{P}$ ensures some requirement $\mathcal{R}$, this analysis is always with respect to a specific model $\mathcal{M}$; namely, we only consider whether $\mathcal{R}$ holds for executions of $\mathcal{P}$ against an adversary $\mathcal{A}$ whose executions satisfy the model (assumptions) $\mathcal{M}$.

CMoSS models capture assumptions on the adversary's capabilities, which include delays, clock bias and 'non-malicious' faults. Requirements and models are independent. The same requirement $\mathcal{R}$ may be satisfied by protocol $\mathcal{P}_1$ under model $\mathcal{M}_1$, or by protocol $\mathcal{P}_2$ under model $\mathcal{M}_2$, facilitating fair comparison of different protocols, since they both meet the same requirement $\mathcal{R}$, just under different models and possibly with different performance. Similarly, the same model $\mathcal{M}$ may be assumed by different protocols $\mathcal{P}_1$ and $\mathcal{P}_2$, possibly designed to satisfy different requirements $\mathcal{R}_1$ and $\mathcal{R}_2$. This modularity allows re-use of models and direct comparison between different protocols for the same requirement. In the rest of this section, we first provide an overview of the execution process, and then formally define these concepts and motivate their key elements.

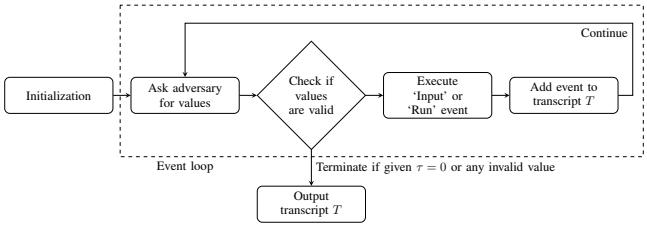

Figure 2. High-level overview of CMoSS's adversary-driven execution process.

Conventions.. Given an array, mapping, or dictionary denoted x, we define $x[y]$ and $x[y, z]$ to return $\perp$, if the corresponding entries (y and y, z) are *not* defined for x. Similarly, for a name-value set $S \equiv \{(N_1, v_1), (N_2, v_2), \ldots\}$, $S[N]$ returns v if $(N, v) \in S$ and there is not $v' \neq v$ s.t. $(N, v') \in S$; otherwise, $S[N] = \perp$. When the names of S or the indices of an array/mapping/dictionary x are strings (field names), we often use *dot notation* to refer to the corresponding value. For example, each event e in an execution transcript T is a name-value set, and we refer to a specific field of e using dot notation, e.g., $e.\tau$ is the time when e occurred; formally, $e.\tau \equiv e[\text{'}\tau\text{'}]$.

2.2. The CMoSS Execution Process

We now describe the CMoSS execution process (Algorithm 1) and its rationale; see a simplified overview in Figure 2. Each CMoSS execution defines a sequence of events. The outcome of the CMoSS execution process is a random variable called the *execution transcript*, denoted $T \leftarrow \mathbf{Exec}^{X}_{\mathcal{A},\mathcal{P}}(1^{\kappa})$, where $\mathcal{P}$ is the executed protocol, $\mathcal{A}$ is an adversary, and 1^{κ} is the security parameter. The transcript T is a set of event entries, where each entry $e \in T$ contains the values from the adversary and from the protocol in a single iteration (event) in the execution. The X input is optional; when used, it defines 'extension' execution operations.

The following discussion is organized around different aspects of the CMoSS execution process: the protocol and subprotocols, entities and instances, the adversary, real-time concurrency using 'Input' and 'Run' events, adding past outputs to inputs and eavesdropping, execution operations, event fields, and informal description of the execution process. §C includes additional details.

2.2.1. Protocol and Subprotocols. To make it easier to define the composition of protocols, CMoSS defines a

Algorithm 1 The CMoSS Adversary-Driven Execution Process $\mathbf{Exec}^{X}_{\mathcal{A},\mathcal{P}}(1^\kappa)$. See notations in Table 2. The bound B, calculated in Line 2, is used to perform bound-checking in Lines 7 and 11 to ensure that the total runtime of the execution remains polynomial. This is essential when analyzing executions involving multiple interacting polynomial-time algorithms; the need to bound the joint runtime was identified in [19] and addressed in [7], [17].

1: $s_{\mathcal{A}} \leftarrow \mathcal{A}[\text{INIT}]\left(1^\kappa\right), s_X \leftarrow X[\text{INIT}]\left(1^\kappa, |s_{\mathcal{A}}|\right)$ // *Initialize state $s_{\mathcal{A}}$ of adversary $\mathcal{A}$ and state s_X of the X operations set*

2: $B \leftarrow \max\{|s_X|, |s_{\mathcal{A}}|, \max\{|\mathcal{P}[sp, \text{'Bound'}](1^\kappa)|\}_{sp\in\mathcal{P}[\cdot]}\}$ // *Set bound B to the maximum of the sizes of s_X, $s_{\mathcal{A}}$, and the maximum size of the output of the 'Bound' operation of any subprotocol of $\mathcal{P}$*

3: $\forall ent, sp, ins \in \{0,1\}^* : sts[ent, sp, ins] \leftarrow \{(\text{'}1^\kappa\text{'}, 1^\kappa)\}$ // *Initialize a state for every possible entity, subprotocol, and instance triple (ent, sp, ins)*

4: $T \leftarrow \emptyset$ // *Initialize the transcript*

5: **loop**

6: $(\tau, vals, EventType) \leftarrow \mathcal{A}[\text{SELECT}]\left(s_{\mathcal{A}}\right)$ // *Let $\mathcal{A}$ select the time, values, and event type*

7: **if** $\neg$VALIDVALS$(T, \tau, vals, EventType, B)$ **then return** T // *End the execution, returning the transcript T, if $(\tau, vals, EventType)$ are not valid (as defined in Alg. 5), e.g., if $\tau \leq 0$*

8: **if** $EventType =$ 'Input' **then**

// *Use the SaveInput function (Alg. 5) to return an event containing the time τ and other values based on $vals$, including the entity, instance, and subprotocol identifiers, the operation name, and the full input (built from the adversary's input and past events' outputs).*

9: $newEvent \leftarrow$ SAVEINPUT$(\tau, T, vals)$

10: **else**

// *Use the* RUNOPR *function (Alg. 7) to execute the operation saved in the event specified in $vals$. This may be an X-operation, an operation of $\mathcal{P}$, or the adversary operation $\mathcal{A}$[RUN]. Return new states and an event containing the time τ and other values based on $vals$ and the outputs of the executed operation, including the entity, subprotocol, and instance identifiers, the operation name, and the output.*

11: $(s_{\mathcal{A}}, s_X, sts, newEvent) \leftarrow$ RUNOPR$(\tau, s_{\mathcal{A}}, s_X, sts, T, vals, B)$

12: **end if**

13: $T \leftarrow T \cup \{newEvent\}$ // *Add event to transcript*

14: **end loop**

protocol $\mathcal{P}$ as a mapping from *pairs* of binary[3] strings (sp, op_name), to algorithms or to $\perp$. We refer to the two strings (sp, op_name) as a *subprotocol* and an *operation name*, respectively, and to the algorithm $\mathcal{P}[sp, op_name]$ as the *algorithm that is mapped to* (sp, op_name). We say that a string $sp \in \{0,1\}^*$ is a *subprotocol of* $\mathcal{P}$ if it is in the set of subprotocols of $\mathcal{P}$, denoted $\mathcal{P}[\cdot]$, where:

$$\mathcal{P}[\cdot] \equiv \left\{ sp \;\middle|\; \begin{array}{l} \exists op_name \in \{0,1\}^* \\ \text{s.t. } \mathcal{P}[sp, op_name] \neq \perp \end{array} \right\}$$

We say that a protocol $\mathcal{P}$ is probabilistic polynomial time (PPT), and write $\mathcal{P} \in PPT$, if all the algorithms mapped by operations of subprotocols of $\mathcal{P}$ are probabilistic polynomial time (PPT) algorithms, i.e.:

$$\mathcal{P} \in PPT \iff \forall sp \in \mathcal{P}[\cdot], op_name \in \{0,1\}^* : \mathcal{P}[sp, op_name] \in PPT \cup \{\perp\}$$

The use of a subprotocol identifiers allows us to compose two (or more) protocols into a composite protocol, even if there are operations with the same name in the protocols being composed. In particular, we often use INIT for the initialization operation; therefore, when a protocol $\mathcal{P}$ is composed of two or more subprotocols, then $\mathcal{P}$ might have multiple INIT algorithms, all with the same operation name (INIT), and each identified using a different subprotocol identifier. For example, the INIT operation of the clock-synchronization (CS) protocol, shown in Algorithm 20, uses the INIT operation of the authenticated request-response (ARR) protocol, shown in Algorithm 22. To refer to the operations of a specific subprotocol we often use the dot notation $sp.op_name$, e.g., CS.INIT and ARR.INIT.

In an execution with protocol $\mathcal{P}$, if a subprotocol sp is invoked which is in $\mathcal{P}[\cdot]$ but $\mathcal{P}[sp, op_name] = \perp$, this means that operation op_name of subprotocol sp is *undefined* in $\mathcal{P}$; invoking an undefined operation of a subprotocol of $\mathcal{P}$ results in invalid values and terminating the execution. Invoking an X operation which is undefined, i.e., $X[op_name] = \perp$, also results in invalid values and terminating the execution.

3. For convenience, we sometimes use character strings; these should be interpreted as their binary representation.

2.2.2. Entities and Instances. The adversary can instruct the execution process to invoke operations of different subprotocols and at different entities. Entities are stateful, with separate state for different subprotocols. Furthermore, an entity may run multiple instances of the same subprotocol, each instance with its own separate state. Namely, when the adversary instructs the execution process to invoke operations, the adversary specifies the entity, the subprotocol and the instance. The execution process maintains a separate state for each (entity, subprotocol, instance) triplet, providing the state as an input when invoking the operation and saving the updated state upon completion of the operation.

To illustrate, our clock synchronization protocol $\mathcal{P}_{\text{UnauthCS}}$ (Algorithm 20) can be analyzed by itself, including only a single subprotocol CS, or as the composition of the $\mathcal{P}_{\text{UnauthCS}}$ protocol and the authenticated request-response protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$, which includes the single subprotocol ARR (Algorithm 22). An entity may run multiple instances of the ARR subprotocol, for example, one used by an instance of the CS subprotocol and one used by an instance of another subprotocol, e.g., DNS. Or, for redundancy, a client may run two instances of CS, each with its own instance of ARR, and each using a different clock server, with its own address and public key.

To support this, states are stored in the execution process using the sts variable, and each state is identified as $sts[ent, sp, ins]$, where ent is an entity identifier, ins is an instance identifier, and sp is a subprotocol identifier.

2.2.3. Adversary. An adversary $\mathcal{A}$ is a mapping from three specific operation names: INIT, SELECT, and RUN, to algorithms. We say that an adversary $\mathcal{A}$ is PPT if its three algorithms are PPT.

The $\mathcal{A}$[INIT] operation is used by the execution process to initialize the adversary before the event loop starts. The execution process invokes the adversary's $\mathcal{A}$[SELECT] operation at the beginning of every iteration of the event loop, so the adversary can decide what to do next. In each event, the values provided by the adversary are verified by the execution process, and if they are invalid (as defined in §C), then the execution terminates and returns the transcript. This gives the adversary the ability to terminate the execution by intentionally providing invalid values.

The execution process allows the adversary to control instances of subprotocols that are not in $\mathcal{P}[\cdot]$, even at benign entities. This approach is motivated by the convenience it provides for secure composition, in particular, to prove Theorem 1. In 'Run' events at instances of subprotocols not in $\mathcal{P}[\cdot]$, the RUNOPR function allows the adversary to decide the state and output of an instance using the $\mathcal{A}$[RUN] operation. That said, the separation of states between different instances ensures that the adversary cannot expose or tamper with the states of instances of subprotocols which are in $\mathcal{P}[\cdot]$. The $\mathcal{A}$[RUN] operation outputs the desired output of the (non-existing) subprotocol. This allows the adversary to simulate subprotocols not implemented by the protocol. For example, in §H.2.1, we describe the implementation of the authenticated communication protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ that relies on a communication subprotocol COM; however, we do not provide an actual implementation of the COM subprotocol. Instead, we let the adversary 'fill in' the missing functionality of COM, which is what happens when an invoked subprotocol is undefined. Then, using the model $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ (discussed in §4.3), we can ensure that the adversary actually behaves as expected, in order to ensure the satisfaction of requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$ by protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$.

2.2.4. Real-time Concurrency Using 'Input' and 'Run' Events. In real systems, operations are not instantaneous, i.e., some time elapses between receiving inputs and producing outputs. This has a significant impact on correctness, efficiency, reliability, and security of distributed systems; see, e.g., the extensive work on timed I/O Automata [21] and secure reactive systems [1]. Therefore, support for real-time concurrency is critical to facilitate analysis of real systems.

CMoSS supports real-time concurrency as follows. At the beginning of each iteration of the event loop, the execution process asks the adversary (via the SELECT operation) to choose between two types of events: (1) *'Input' events*, where the adversary specifies inputs for *future* events, and (2) *'Run' events*, where protocol instances or X operations are run and return outputs. The execution transcript is used to store values provided in 'Input' events. After the adversary saves the input, it can instruct the execution process to give the input to a (protocol or adversary-controlled) instance or to an X-operation in a 'Run' event (which is also recorded in execution transcript). Specifically, in a 'Run' event, the adversary outputs the time τ_{Input} of a past 'Input' event. Not all 'Input' events may have corresponding 'Run' events by the end of an execution, since an execution may end at the beginning of any iteration. The adversary specifies the real time of each event (and may specify any local clock values as part of the inputs to protocol instances); the only restriction is that the adversary must give strictly increasing real time values for consecutive events. There can be multiple input events at the same instance before an output event. Thus, CMoSS supports concurrency both at different instances and at a single instance.

2.2.5. Adding Past Outputs to Inputs and Eavesdropping. Inputs from 'Input' events are used in 'Run' events as described, above. Additionally, outputs from 'Run' events can be used as part of inputs in 'Input' events. In particular, in 'Input' events, the adversary outputs the direct input inp, and the past outputs set *pastouts* (containing names and times of fields outputted in past 'Run' events). The full input $fullinp$ is constructed using inp, *pastouts*, and past events recorded in the transcript, and is saved in the new event structure to the transcript.

The adversary can use the past outputs mechanism and the $\mathcal{A}$[RUN] operation to eavesdrop on outputs of subprotocols which are in $\mathcal{P}[\cdot]$. In particular, the adversary can choose a subprotocol which is not in $\mathcal{P}[\cdot]$ (i.e., choose do an event at an adversarial instance), and it can add past output fields of instances of subprotocols in $\mathcal{P}[\cdot]$ to the input to the adversarial instance. This can also be used by the adversary

to learn unknown field names. For example, the adversary may eavesdrop on a known field name '$Params$' (used in the ASK convention described in §4.3), which contains the names of other relevant output fields. Then, in another event, the adversary can eavesdrop on these output fields as well. Models (see §2.4) can be used to limit or prevent the adversary's eavesdropping capabilities.

2.2.6. Execution Operations. The CMoSS execution process allows definition of many types of requirements and models, e.g., MitM and eavesdropper adversaries. However, some requirements and models require additional functionalities, not provided by the 'base' execution process. We define these using the (optional) input X to the execution process $\mathbf{Exec}^{X}_{\mathcal{A},\mathcal{P}}$. The X input is a mapping from binary strings[4] to algorithms, which we call *execution operations (X-operations)*.

For convenience, we define a default execution operations mapping X^*, which contains the two execution operations described below. Specifically, the default execution operations mapping X^* is defined as:

$$X^* \equiv \left\{ \text{ GET-STATE, SET-STATE } \right\}$$

The two corruption execution operations GET-STATE and SET-STATE, defined in §D, allow modeling adversaries that can access (read and/or modify) the states of entities. Depending on the model, this access could be allowed for the entire execution or temporarily, at all entities or at only some entities. This allows expressing classical adversary models such as *honest-but-curious adversaries* [14], *threshold-cryptography* [10], *self-stabilization* [11], *proactive cryptography* [8], *leakage-resiliency* [12] and more.

2.3. Specifications

CMoSS uses *predicates over execution transcripts* to define both *models* (assumptions) and *requirements*. Let us first define predicate satisfaction and failure probability.

Definition 1 (Predicate satisfaction and failure probability)**.** *Let T be an execution transcript and π be a predicate over transcripts; we say that T* satisfies *π if $\pi(T) = \top$.*

We define the failure probability of *π for executions with adversary $\mathcal{A}$, protocol $\mathcal{P}$, execution operations X, and security parameter κ, denoted $\epsilon^{\pi,X}_{\mathcal{A},\mathcal{P}}(\kappa)$, as:*

$$\epsilon^{\pi,X}_{\mathcal{A},\mathcal{P}}(\kappa) \stackrel{def}{=} \Pr\left[\begin{array}{l} \pi(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}^{X}_{\mathcal{A},\mathcal{P}}(1^\kappa) \end{array} \right] \quad (1)$$

We next define *model specifications* and *requirement specifications*, also called *property specifications*.

Model specifications are used to define *models* (§2.4), which restrict what the adversary can do. A model specification $\mathcal{M}$ is a triple $\mathcal{M} \equiv (\mathcal{M}.\pi, \mathcal{M}.\beta, \mathcal{M}.Sp)$, where $\mathcal{M}.\pi$ is a predicate that should hold over execution transcripts, $\mathcal{M}.\beta$ is the allowed failure probability for $\mathcal{M}.\pi$ (often, $\mathcal{M}.\beta = 0$), and $\mathcal{M}.Sp$ is the set of subprotocols which the predicate $\mathcal{M}.\pi$ concerns. $\mathcal{M}.Sp$ should be $\emptyset$ if the model specification is not subprotocol-specific. Otherwise, if the protocol includes any of the subprotocols in $\mathcal{M}.Sp$, then we say that $\mathcal{M}$ holds, regardless of the actual failure probability of $\mathcal{M}.\pi$, since the adversary cannot control subprotocols which are part of the protocol and consequently may not be able to ensure that $\mathcal{M}.\pi$ holds in executions with such a protocol.

A model specification, and similarly a requirement specification, may require either *unconditional satisfaction* or *asymptotic satisfaction*. In the case of unconditional satisfaction, the failure probability cannot exceed the allowed failure probability. In the case of asymptotic satisfaction, the failure probability can exceed the allowed failure probability by a negligible function computed over the security parameter input to the execution, which we denote by κ.

Definition 2 (Model specification and its satisfaction)**.** *A* ***model specification*** *$\mathcal{M}$ is a triple $\mathcal{M} \equiv (\mathcal{M}.\pi, \mathcal{M}.\beta, \mathcal{M}.Sp)$ where $\mathcal{M}.\pi$ is a predicate over execution transcripts, and $0 \leq \mathcal{M}.\beta < 1$. We say that a model specification $\mathcal{M} \equiv (\mathcal{M}.\pi, \mathcal{M}.\beta, \mathcal{M}.Sp)$* ***holds unconditionally*** *for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, denoted as $(\mathcal{A},\mathcal{P}) \models_{X,\infty} \mathcal{M}$, if there is a subprotocol that is in both $\mathcal{M}.Sp$ and in $\mathcal{P}[\cdot]$, or the failure probability of $\mathcal{M}.\pi$ is at most $\mathcal{M}.\beta$. Namely:*

$$(\mathcal{A},\mathcal{P}) \models_{X,\infty} \mathcal{M} \Leftrightarrow \left[\begin{array}{l} \mathcal{M}.Sp \cap \mathcal{P}[\cdot] \neq \emptyset \\ \vee\ \forall \kappa \in \mathbb{N} : \epsilon^{\mathcal{M}.\pi,X}_{\mathcal{A},\mathcal{P}}(\kappa) \leq \mathcal{M}.\beta \end{array} \right]$$

We say that a specification $\mathcal{M} \equiv (\mathcal{M}.\pi, \mathcal{M}.\beta, \mathcal{M}.Sp)$ ***holds asymptotically*** *for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, denoted as $(\mathcal{A},\mathcal{P}) \models_{X,\text{POLY}} \mathcal{M}$, if there is a subprotocol that is in both $\mathcal{M}.Sp$ and in $\mathcal{P}[\cdot]$, or the failure probability of $\mathcal{M}.\pi$ is at most $\mathcal{M}.\beta + Negl(1^\kappa)$. Namely:*

$$(\mathcal{A},\mathcal{P}) \models_{X,\text{POLY}} \mathcal{M} \Leftrightarrow \left[\begin{array}{l} \mathcal{M}.Sp \cap \mathcal{P}[\cdot] \neq \emptyset \\ \vee\ \forall \kappa \in \mathbb{N} : \epsilon^{\mathcal{M}.\pi,X}_{\mathcal{A},\mathcal{P}}(\kappa) \leq \mathcal{M}.\beta + Negl(1^\kappa) \end{array} \right]$$

Requirement specifications are used to define *requirements* (§2.4), which describe what protocols should do. The terms *property specifications* and *properties*, respectively, can also be used when describing protocols' properties which are not necessarily required. A requirement specification $\mathcal{R}$ is a pair $\mathcal{R} \equiv (\mathcal{R}.\pi, \mathcal{R}.\beta)$, where $\mathcal{R}.\pi$ is a predicate that should hold over execution transcripts and $\mathcal{R}.\beta$ is the allowed failure probability for $\mathcal{R}.\pi$ (often, $\mathcal{R}.\beta = 0$).

Cryptographic requirements may be ensured asymptotically, while non-cryptographic distributed computing requirements may be ensured unconditionally. For many requirements, ensuring unconditional satisfaction is hard or impossible, however, assuming unconditional satisfaction can be convenient for writing security proofs, even if the final satisfaction is asymptotic.

Definition 3 (Requirement specification and its satisfaction)**.** *A* ***requirement specification*** *$\mathcal{R}$ is a pair $\mathcal{R} \equiv (\mathcal{R}.\pi, \mathcal{R}.\beta)$*

4. Character strings should be interpreted as their binary representation.

where $\mathcal{R}.\pi$ is a predicate over execution transcripts, and $0 \leq \mathcal{R}.\beta < 1$. We say that a requirement specification $\mathcal{R} \equiv (\mathcal{R}.\pi, \mathcal{R}.\beta)$ holds unconditionally for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, denoted as $(\mathcal{A}, \mathcal{P}) \models_{X,\infty} \mathcal{R}$, if the failure probability of $\mathcal{R}.\pi$ is at most $\mathcal{R}.\beta$. Namely:

$$(\mathcal{A}, \mathcal{P}) \models_{X,\infty} \mathcal{R} \Leftrightarrow \forall \kappa \in \mathbb{N} : \epsilon_{\mathcal{A},\mathcal{P}}^{\mathcal{R}.\pi,X}(\kappa) \leq \mathcal{R}.\beta$$

We say that a specification $\mathcal{R} \equiv (\mathcal{R}.\pi, \mathcal{R}.\beta)$ holds asymptotically for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, denoted as $(\mathcal{A}, \mathcal{P}) \models_{X,\text{POLY}} \mathcal{R}$, if the failure probability of $\mathcal{R}.\pi$ is at most $\mathcal{R}.\beta + Negl(1^\kappa)$. Namely:

$$(\mathcal{A}, \mathcal{P}) \models_{X,\text{POLY}} \mathcal{R} \Leftrightarrow \forall \kappa \in \mathbb{N} : \epsilon_{\mathcal{A},\mathcal{P}}^{\mathcal{R}.\pi,X}(\kappa) \leq \mathcal{R}.\beta + Negl(1^\kappa)$$

Observation 1. *For any model specification $\mathcal{M}$ and requirement specification $\mathcal{R}$:*

1) $(\mathcal{A}, \mathcal{P}) \models_{X,\infty} \mathcal{M} \Rightarrow (\mathcal{A}, \mathcal{P}) \models_{X,\text{POLY}} \mathcal{M}$

2) $(\mathcal{A}, \mathcal{P}) \models_{X,\infty} \mathcal{R} \Rightarrow (\mathcal{A}, \mathcal{P}) \models_{X,\text{POLY}} \mathcal{R}$

2.4. Models and Requirements

As mentioned in the previous section, models restrict what the adversary can do, while requirements, also called properties, describe what protocols do or should do. In this section, we formally define models, requirements, and their satisfaction by adversaries or protocols, respectively.

A model $\mathcal{M} \equiv \{\mathcal{M}_i\}_{i \in \{1,\ldots,|\mathcal{M}|\}}$ is a set of model specifications (§2.3). A model restricts the executions of an adversary $\mathcal{A}$ with arbitrary protocols which do not include the subprotocols in the sets $\mathcal{M}_i.Sp$. Each predicate $\mathcal{M}_i.\pi$ should describe a property which an adversary is able to ensure for every possible protocol which does not include any subprotocol in $\mathcal{M}_i.Sp$, even unusual protocols; otherwise, no adversary satisfies the model.

A requirement, or property, $\mathcal{R} \equiv \{\mathcal{R}_i\}_{i \in \{1,\ldots,|\mathcal{R}|\}}$ is a set of requirement specifications (§2.3). A requirement describes the expected properties of executions of a protocol $\mathcal{P}$ with any adversary $\mathcal{A}$ that satisfies a given model $\mathcal{M}$. Note that any model specification $\mathcal{M} \in \mathcal{M}$ where $\mathcal{M}.Sp$ includes a subprotocol in $\mathcal{P}[\cdot]$ is unnecessary and can be removed from the model, because it does not restrict adversaries in executions with $\mathcal{P}$.

If a model includes only one model specification, we may abuse notation and use the same notation $\mathcal{M}$ for both the model and the model specification. Similarly, if a requirement includes only one requirement specification, we may abuse notation and use the same notation $\mathcal{R}$ for both the requirement and the requirement specification.

We begin with the definition of models and define what it means for an adversary $\mathcal{A}$ to satisfy a model.

Definition 4 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$ using execution operations X)**.** *Let $\mathcal{A}$ be a PPT adversary, X be execution operations, and $\mathcal{M} \equiv \{\mathcal{M}_i\}_{i \in \{1,\ldots,|\mathcal{M}|\}}$ be a set of model specifications. We say that $\mathcal{A}$ unconditionally (or asymptotically) satisfies model $\mathcal{M}$ using execution operations X, denoted $\mathcal{A} \models_{X,\infty} \mathcal{M}$ (or $\mathcal{A} \models_{X,\text{POLY}} \mathcal{M}$), if for every PPT protocol $\mathcal{P}$, every model specification $\mathcal{M}_i \in \mathcal{M}$ holds unconditionally (or asymptotically) for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, i.e., for* TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\mathcal{A} \models_{X,\text{TYPE}} \mathcal{M} \Leftrightarrow \forall \mathcal{P} \in PPT, \mathcal{M}_i \in \mathcal{M} : (\mathcal{A}, \mathcal{P}) \models_{X,\text{TYPE}} \mathcal{M}_i$$

We next present an important example: the $\mathcal{M}_{\Delta_{io}}^{\text{IO}}$ model, which bounds the delay between an input event and the corresponding output event. Note that $\mathcal{M}_{\Delta_{io}}^{\text{IO}}$ has a parameter, Δ_{io}, which is the maximal delay. Many models and requirements have parameters.

Example 1 (Bounded input-output delay model)**.** *The CMoSS execution process allows delay for the execution of an event, i.e., it separates the event where the inputs are provided, from the event where the operation is run and the output values are computed and returned to the adversary. In the time between the input and output events, there may be other (input and/or output) events at the same instance or at other instances. A model may enforce bounds on the delay between input and output of an event.*

The input-output delay model (Algorithm 2) bounds the time between an input event and the corresponding output event. This model, or a similar one, is assumed whenever a protocol should satisfy a requirement that requires some output within bounded time. The input-output delay model has one model specification $\mathcal{M}_{\Delta_{io}}^{IO} \equiv (\mathcal{M}_{\Delta_{io}}^{IO}.\pi, \mathcal{M}_{\Delta_{io}}^{IO}.\beta, \mathcal{M}_{\Delta_{io}}^{IO}.Sp)$. Predicate $\mathcal{M}_{\Delta_{io}}^{IO}.\pi$ ensures that the delay between every input event and its corresponding output event does not exceed the bound Δ_{io}, specified as a parameter of $\mathcal{M}_{\Delta_{io}}^{IO}.\pi$, unless the execution terminates within Δ_{io} time of the input event (in which case, the model holds in a trivial way). As in many model and requirement specifications, the allowed failure probability $\mathcal{M}_{\Delta_{io}}^{IO}.\beta$ is 0. The subprotocol set $\mathcal{M}_{\Delta_{io}}^{IO}.Sp$ is empty, because the model is not subprotocol-specific.

□

The following simple observation allows us to use models in a modular way and is often useful when there are multiple assumptions.

Observation 2. $\forall \mathcal{M}, \mathcal{M}'$, *for* TYPE $\in \{\infty, \text{POLY}\}$:

$$\mathcal{A} \models_{X,\text{TYPE}} (\mathcal{M} \cup \mathcal{M}') \iff \left(\mathcal{A} \models_{X,\text{TYPE}} \mathcal{M} \wedge \mathcal{A} \models_{X,\text{TYPE}} \mathcal{M}'\right)$$

Next, we define requirements and what it means for a protocol $\mathcal{P}$ to satisfy a requirement assuming a model.

Definition 5 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$ using execution operations X)**.** *Let $\mathcal{P}$ be a PPT protocol, X be execution operations, $\mathcal{M}$ be a set of model specifications, and $\mathcal{R} \equiv \{\mathcal{R}_i\}_{i \in \{1,\ldots,|\mathcal{R}|\}}$ be a set*

Algorithm 2 Input-output delay model, with model specification $\mathcal{M}^{\text{IO}}_{\Delta_{io}} \equiv (\mathcal{M}^{\text{IO}}_{\Delta_{io}}.\pi, \mathcal{M}^{\text{IO}}_{\Delta_{io}}.\beta, \mathcal{M}^{\text{IO}}_{\Delta_{io}}.Sp)$

1: **Allowed failure probability** $\mathcal{M}^{\text{IO}}_{\Delta_{io}}.\beta \equiv 0$
2: **Subprotocols** $\mathcal{M}^{\text{IO}}_{\Delta_{io}}.Sp \equiv \emptyset$
3: **Model predicate** $\mathcal{M}^{\text{IO}}_{\Delta_{io}}.\pi(T)$
4: For every input event $e \in T$ ($e.EventType =$ 'Input'), except if e occurs in the last Δ_{io} time of the execution, there is a later output event e' ($e'.EventType =$ 'Run') at the same instance $(e.ent, e.sp, e.ins)$, such that:
5: - e' is at most Δ_{io} time after e, i.e.: $e'.\tau \leq e.\tau + \Delta_{io}$,
6: - and e' uses the input from event e, i.e.: $e'.\tau_{\text{Input}} = e.\tau$.
7: **end model predicate**

of requirement specifications. We say that $\mathcal{P}$ unconditionally (or asymptotically) satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$ using execution operations X, denoted $\mathcal{P} \models^{\mathcal{M}}_{X,\infty} \mathcal{R}$ (or $\mathcal{P} \models^{\mathcal{M}}_{X,\text{POLY}} \mathcal{R}$), if for every PPT adversary $\mathcal{A}$ that unconditionally (or asymptotically) satisfies model $\mathcal{M}$, every requirement specification $\mathcal{R}_i \in \mathcal{R}$ holds unconditionally (or asymptotically) for adversary $\mathcal{A}$, protocol $\mathcal{P}$, and execution operations X, i.e., for TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\mathcal{P} \models^{\mathcal{M}}_{X,\text{TYPE}} \mathcal{R} \Leftrightarrow \forall \mathcal{A} \in PPT \text{ s.t. } \mathcal{A} \models_{X,\text{TYPE}} \mathcal{M}, \mathcal{R}_i \in \mathcal{R} : (\mathcal{A}, \mathcal{P}) \models_{X,\text{TYPE}} \mathcal{R}_i$$

Observation 3. $\forall \mathcal{M}' \subseteq \mathcal{M}, \mathcal{R} = \mathcal{R}' \cup \mathcal{R}''$, *for* TYPE $\in \{\infty, \text{POLY}\}$:

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

Example 2 (Server authentication requirement)**.** *The server authentication requirement (Algorithm 3) is part of the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}$ defined in §H.1.4. The server authentication requirement has one requirement specification $\mathcal{R}^{ARR}_{\text{SRVAUTH}} \equiv (\mathcal{R}^{ARR}_{\text{SRVAUTH}}.\pi, \mathcal{R}^{ARR}_{\text{SRVAUTH}}.\beta)$. The predicate $\mathcal{R}^{ARR}_{\text{SRVAUTH}}.\pi$ ensures the authenticity of server responses outputted by clients. More precisely, if a client instance of subprotocol ARR outputs a (server) response with an identifier, then previously the client was invoked to send a request with this identifier to the server, and the server was invoked to send this response to the client. The allowed failure probability $\mathcal{R}^{ARR}_{\text{SRVAUTH}}.\beta$ is 0.*

The server authentication requirement $\mathcal{R}^{ARR}_{\text{SRVAUTH}}$ can be ensured by different protocols, possibly assuming different models. In §H.2.1, we show a simple protocol $\mathcal{P}^{\mathcal{S}}_{ARR}$ that asymptotically ensures $\mathcal{R}^{ARR}_{\text{SRVAUTH}}$ against Man-in-the-Middle (MitM) *adversaries. To simplify our theorem, so that we assume the same model for the entire authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}$, we assume a (stronger) model which makes the adversary weaker than MitM, i.e., off-path and eavesdropping, where the adversary can send spoofed packets and can eavesdrop on messages sent by benign entities, but cannot drop benign packets. However, $\mathcal{P}^{\mathcal{S}}_{ARR}$ would also asymptotically ensure $\mathcal{R}^{ARR}_{\text{SRVAUTH}}$ against MitM adversaries. A weaker or different protocol could ensure the same requirement assuming a more restrictive (stronger) adversary model, e.g., off-path-only or eavesdropping-only.* □

Algorithm 3 Server authentication requirement, with req. specification $\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}} \equiv (\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\pi, \mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\beta)$

1: **Allowed failure probability** $\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\beta \equiv 0$
2: **Requirement predicate** $\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\pi(T)$
3: Suppose that (1) instance ins_S of subprotocol ARR, in benign entity ent_S, asked to invoke operation ARR-INITRESPSERVER with parameters ADDR:$addr$ and PK:PK, (2) instance ins_C, running in a benign entity ent_C, has an ARR.INIT event with parameters ROLE:CLIENT, SRVADDR:$addr$ and SRVPK:PK, (3) instance ins_C is assigned address $ClAddr$, and (4) instance ins_C asks to invoke ARR-RCVRESP with parameters RESP:$resp$ and REQID:$reqID$. Then:
4: (1) There was an ARR.SENDREQ event in ins_C with parameters REQ; denote the value as req, and REQID; denote the value as $reqID$.
5: (2) There was an ARR.SENDRESP event at ins_S, with parameters RESP:$resp$, REQID:$reqID$, and CLADDR:$ClAddr$.
6: **end requirement predicate**

We state two properties in Observation 4 which capture standard weakening between unconditional and asymptotic satisfaction notions. Namely, if an adversary $\mathcal{A}$ unconditionally satisfies model $\mathcal{M}$, then $\mathcal{A}$ also asymptotically satisfies $\mathcal{M}$, and if a protocol $\mathcal{P}$ unconditionally satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$, then $\mathcal{P}$ also asymptotically satisfies $\mathcal{R}$ assuming $\mathcal{M}$.

Observation 4. *Let $\mathcal{P}$ be a PPT protocol, X be execution operations, $\mathcal{M}$ be a model, and $\mathcal{R}$ be a requirement. Then:*

1) $\mathcal{A} \models_{X,\infty} \mathcal{M} \Rightarrow \mathcal{A} \models_{X,\text{POLY}} \mathcal{M}$
2) $\mathcal{P} \models^{\mathcal{M}}_{X,\infty} \mathcal{R} \Rightarrow \mathcal{P} \models^{\mathcal{M}}_{X,\text{POLY}} \mathcal{R}$

3. Composition of Protocols in CMoSS

CMoSS enables *modular protocol design*, i.e., satisfying requirements through the *composition of two or more protocols*. In this section, we explain how protocols are composed (§3.1), present lemmas describing properties of composition (§3.2), and show that *composition preserves requirement satisfaction*, i.e., the protocol which is the composition of protocol $\mathcal{P}$ with protocol $\mathcal{P}'$ satisfies any requirement already satisfied by $\mathcal{P}$ (§3.3 and Theorem 1).

3.1. Defining Composition of Protocols

In CMoSS, a protocol is a mapping from pairs (sp, op_name) to algorithms. Therefore, intuitively - and often - the composition of two protocols $\mathcal{P}$ and $\mathcal{P}'$ is simply the union of these mappings. However, what if $\mathcal{P}$ and $\mathcal{P}'$ both map the same pair (sp, op_name) but to different algorithms? Furthermore, the CMoSS execution process maintains a separate state for each triple (ent, ins, sp), i.e., entity, instance and subprotocol. Therefore, it suffices for $\mathcal{P}$ and $\mathcal{P}'$ to include the same subprotocol sp, with different operations op_name, op_name', to allow possible unintended interactions between the use of the state of an instance, when the state $s[ent, ins, sp]$ is accessed or modified by algorithm $\mathcal{P}[sp, op_name]$ and by algorithm $\mathcal{P}'[sp, op_name']$.

To prevent such unintended interactions between protocols, CMoSS defines composition so that each subprotocol sp is inherited entirely from one of the protocols being composed. In particular, when protocol $\mathcal{P}$ is composed with protocol $\mathcal{P}'$, then the new protocol, denoted $\mathcal{P}'' \equiv \mathcal{P} \cup \mathcal{P}'$, includes all subprotocols and their operations from $\mathcal{P}$, as well as the subprotocols of $\mathcal{P}'$ that are *not already included in* $\mathcal{P}$. That is, if both $\mathcal{P}$ and $\mathcal{P}'$ define the same subprotocol sp, then the composite protocol $\mathcal{P} \cup \mathcal{P}'$ retains only the operations of sp defined in $\mathcal{P}$. For any subprotocol included in only one of $\mathcal{P}$ and $\mathcal{P}'$, the composite protocol inherits all operations of that subprotocol from $\mathcal{P}$ or $\mathcal{P}'$, respectively.

Specifically, this means that even if a subprotocol sp exists in both $\mathcal{P}$ and $\mathcal{P}'$ but op_name only exists in $\mathcal{P}'$ for that subprotocol, i.e., $\mathcal{P}[sp, op_name] = \perp$ but $\mathcal{P}'[sp, op_name] \neq \perp$, then $(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = \perp$. This is a subtle but important point, and it is essential for ensuring that requirements satisfied by $\mathcal{P}$ are preserved after composition, as shown in §3.3.

In Definition 6, we define, given any two protocols $\mathcal{P}$ and $\mathcal{P}'$, a protocol which we call the *composition of* $\mathcal{P}$ *with* $\mathcal{P}'$, and denote this protocol as $\mathcal{P} \cup \mathcal{P}'$.

Definition 6 (Composition of protocols)**.** *Let $\mathcal{P}$, $\mathcal{P}'$ be protocols. We define the* composition *of $\mathcal{P}$ with $\mathcal{P}'$, denoted $\mathcal{P} \cup \mathcal{P}'$, as follows:*

$$(\mathcal{P} \cup \mathcal{P}')[sp, op_name] \equiv \begin{cases} \mathcal{P}[sp, op_name] \text{ if } sp \in \mathcal{P}[\cdot] \\ \mathcal{P}'[sp, op_name] \text{otherwise} \end{cases}$$

Definition 6 only defines $\mathcal{P} \cup \mathcal{P}'$, the composition of $\mathcal{P}$ with $\mathcal{P}'$; it does not guarantee any properties of the resulting composite protocol $\mathcal{P} \cup \mathcal{P}'$. In §3.2 and §3.3, we observe several properties of the composition of protocols.

3.2. Composition Properties

We now present several observations about composition, which are formalized in the corresponding Lemmas 1 to 4 (for the proofs, see §E.1). We begin with the observation that composition is associative, a property that follows directly from Definition 6.

Lemma 1 (Composition is associative)**.** *For any three protocols $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2$: $\mathcal{P}_2 \cup (\mathcal{P}_1 \cup \mathcal{P}_0) = (\mathcal{P}_2 \cup \mathcal{P}_1) \cup \mathcal{P}_0$.*

Next, we observe that composition is *not* commutative. Consider two protocols $\mathcal{P}$ and $\mathcal{P}'$ that both include a subprotocol sp, but map operation op_name of subprotocol sp to different algorithms. In this case, the composite protocol $\mathcal{P} \cup \mathcal{P}'$ will define (sp, op_name) as in $\mathcal{P}$, while the composed protocol $\mathcal{P}' \cup \mathcal{P}$ will define (sp, op_name) as in $\mathcal{P}'$.

Lemma 2 (Composition is not commutative)**.** *There exist two protocols $\mathcal{P}, \mathcal{P}'$ such that: $\mathcal{P} \cup \mathcal{P}' \neq \mathcal{P}' \cup \mathcal{P}$.*

Composition of protocols is not commutative because protocols may have *conflicts*. Two protocols are said to have a conflict if there exists a subprotocol that is included in both protocols and yet some operation name of this subprotocol is defined in one protocol but is either defined differently or undefined in the other protocol.[5]

Definition 7 (Non-conflicting protocols)**.** *Let $\{\mathcal{P}_i\}$ be a set of protocols. We say that the protocols in $\{\mathcal{P}_i\}$ are* non-conflicting *if the following holds:*

$$\nexists \mathcal{P}, \mathcal{P}' \in \{\mathcal{P}_i\}, sp \in \mathcal{P}[\cdot] \cap \mathcal{P}'[\cdot], op_name \in \{0,1\}^* \textbf{ s.t. :}$$
$$\mathcal{P}[sp, op_name] \neq \mathcal{P}'[sp, op_name]$$

We next observe that if two protocols $\mathcal{P}$ and $\mathcal{P}'$ are non-conflicting, then changing the order of composition does not change the resulting composite protocol. Thus, while the composition of *any* two protocols is *not* commutative, composition between two *non-conflicting* protocols *is* commutative.

Lemma 3 (Composition of non-conflicting protocols is commutative)**.** *For any non-conflicting protocols $\mathcal{P}$ and $\mathcal{P}'$ holds: $\mathcal{P} \cup \mathcal{P}' = \mathcal{P}' \cup \mathcal{P}$.*

Furthermore, we observe that for two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ which are non-conflicting with a protocol $\mathcal{P}_0$, the composite protocol $\mathcal{P}_1 \cup \mathcal{P}_2$ is also non-conflicting with $\mathcal{P}_0$.

Lemma 4 (Composition preserves the non-conflicting property)**.** *Let $\{\mathcal{P}_i\}_{i=0}^2$ be a set of protocols such that $\mathcal{P}_0, \mathcal{P}_1$ are non-conflicting and $\mathcal{P}_0, \mathcal{P}_2$ are non-conflicting, and let $\mathcal{P}_{12} \equiv \mathcal{P}_1 \cup \mathcal{P}_2$. Then $\mathcal{P}_0, \mathcal{P}_{12}$ are also non-conflicting.*

3.3. Composition Preserves Requirement Satisfaction

Consider protocols $\mathcal{P}$ and $\mathcal{P}'$, and let $\mathcal{P}_C \equiv \mathcal{P} \cup \mathcal{P}'$. In Theorem 1 below, we prove that every requirement $\mathcal{R}$ satisfied by $\mathcal{P}$ (under model $\mathcal{M}$), would also be satisfied by $\mathcal{P}_C$; namely, composing $\mathcal{P}$ with another protocol ($\mathcal{P}'$) would preserve the satisfaction of requirements. In the special

5. The definition allows $\mathcal{P}$ and $\mathcal{P}'$ to include subprotocols which are exactly the same, and subprotocols which are only included in one of $\mathcal{P}$ and $\mathcal{P}'$. This is useful when we want to compose two protocols which both include a subprotocol which is exactly the same, e.g., both protocols include the same communication subprotocol.

Algorithm 4 Adversary $\mathrm{ConstrAdv}(\mathcal{P}', \mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'})$

procedure $\mathrm{ConstrAdv}(\mathcal{P}', \mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'})[op_{\mathcal{A}}](inputs)
  // If the adversary is invoked to run an operation of a subprotocol of $\mathcal{P}'$, then run $\mathcal{P}'$
  if $op_{\mathcal{A}} = \textsc{Run} \wedge sp \in \mathcal{P}'[\cdot]$ **then**
    $(sp, op_name, fullinp, s, s_{\mathcal{A}}) \leftarrow inputs$
    $(out, s) \leftarrow \mathcal{P}'[sp, op_name](fullinp, s)$
    // Return output of $\mathcal{P}'$ and unmodified state $s_{\mathcal{A}}$
    return $(out, s, s_{\mathcal{A}})$
  end if
  // Otherwise, run $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}$ and return its outputs
  return $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}[op_name_{\mathcal{A}}](inputs)$
end procedure

case where $\mathcal{P}$ and $\mathcal{P}'$ are non-conflicting we have that $\mathcal{P}_C \equiv \mathcal{P} \cup\!\!\cup \mathcal{P}' = \mathcal{P}' \cup\!\!\cup \mathcal{P}$, which basically allows *parallel compositions*, as in [13]. Theorem 1 is also a crucial step towards using *black-box compositions* for modular design of protocols, as we do in the next section.

Let us first give some intuition. From Definition 6, for any subprotocol sp which is included in $\mathcal{P}$, all operations with subprotocol sp are defined the same way in $\mathcal{P}$ as in the composition of $\mathcal{P}$ with any protocol $\mathcal{P}'$, i.e., $\mathcal{P}[sp, op_name] = (\mathcal{P} \cup\!\!\cup \mathcal{P}')[sp, op_name]$; informally, we can say that $\mathcal{P} \cup\!\!\cup \mathcal{P}'$ 'contains' $\mathcal{P}$. Therefore, given any adversary $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}$ against $\mathcal{P} \cup\!\!\cup \mathcal{P}'$, we can construct an adversary $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ that will yield the same distribution of results because $\mathcal{P} \cup\!\!\cup \mathcal{P}'$ 'contains' $\mathcal{P}$; the construction is defined in Algorithm 4. Intuitively, $\mathcal{A}_{\mathcal{P}}$ runs 'internally' protocol $\mathcal{P}'$ whenever a subprotocol of $\mathcal{P}'$ is run, and otherwise, $\mathcal{A}_{\mathcal{P}}$ runs 'internally' adversary $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}$. Specifically, we define $\mathcal{A}_{\mathcal{P}} = \mathrm{ConstrAdv}(\mathcal{P}', \mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'})$ as in Algorithm 4.

Before we present the theorem formally, we first present the following lemma, which is used by the proof of the theorem. The proof of this lemma and the proof of the theorem are in §E.1.

Lemma 5. *Let $\mathcal{P}'$ be a PPT protocol and $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}$ be a PPT adversary. Let $\mathcal{A}_{\mathcal{P}} = \mathrm{ConstrAdv}(\mathcal{P}', \mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'})$ defined in Algorithm 4; then $\mathcal{A}_{\mathcal{P}}$ is a PPT adversary s.t. for any PPT protocol $\mathcal{P}$, the distribution of transcripts of executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ is identical to the distribution of transcripts of executions of $\mathcal{A}_{\mathcal{P} \cup\!\!\cup \mathcal{P}'}$ against $(\mathcal{P} \cup\!\!\cup \mathcal{P}')$.*

Theorem 1 (Composition preserves requirement satisfaction)**.** *Let $\mathcal{P}$ be a PPT protocol and $\mathcal{R}$ be a requirement. Then, for any model $\mathcal{M}$, any execution operations X, and any PPT protocol $\mathcal{P}'$, for $\textsc{type} \in \{\infty, \textsc{Poly}\}$, holds:*

$$\mathcal{P} \models^{\mathcal{M}}_{X,\textsc{type}} \mathcal{R} \implies (\mathcal{P} \cup\!\!\cup \mathcal{P}') \models^{\mathcal{M}}_{X,\textsc{type}} \mathcal{R} \qquad (2)$$

4. Modularity via Blackbox Compositions

We now explore an important use of composition: *modular, top-down design of protocols using black-box requirements.* In many protocols, a higher-level protocol needs only a specific guarantee (requirement) from a lower-level protocol, not its implementation. For example, a protocol that relies on clock synchronization can be analyzed using only the clock-synchronization guarantee it needs; the particular clock-synchronization protocol can then be chosen and analyzed separately. This modularity enables simpler and more maintainable protocol specifications, facilitates protocol reuse, allows the same low-level protocol to support multiple high-level protocols, and lets a high-level protocol operate under different models by choosing different lower-level protocols.

In §4.1, we define *black-box requirement satisfaction*, where a protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$, assuming $\mathcal{P}$ can use an arbitrary *black-box protocol* $\mathcal{P}'$ which satisfies some other ('black-box') requirement $\mathcal{R}'$. In §4.2, we present the *black-box composition lemmas*, which extend this top-down design methodology to support multiple layers of top-down design. In §4.3, we present the *ask convention and model*, which support interaction between instances of subprotocols.

4.1. Black-box Requirements

So far, we have considered *monolithic* protocols, i.e., protocols which ensure their requirements without a need for composition with other protocols. Using Definition 5, we are able to specify requirements for monolithic protocols. To support modularity of design, however, we want to be able to specify requirements for *hybrid* protocols, i.e., protocols which require composition with other protocols to ensure their requirements, as well.

We now formalize the *black-box requirement approach*. If a protocol $\mathcal{P}$ *assumes a black-box satisfying* $\mathcal{R}'$, this means that $\mathcal{P}$ depends on requirement $\mathcal{R}'$ being satisfied, without fixing which protocol satisfies it. The requirement $\mathcal{R}'$ can be satisfied by $\mathcal{P}$ itself or using another protocol $\mathcal{P}'$ composed with $\mathcal{P}$. To capture this, we extend Definition 5 to include a black-box requirement.

Definition 8 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$, and a black-box satisfying $\mathcal{R}'$ using execution operations X)**.** *Let $\mathcal{P}$ be a PPT protocol, X be execution operations, $\mathcal{M}$ be a set of model specifications, and $\mathcal{R}$, $\mathcal{R}'$ be sets of requirement specifications. We say that $\mathcal{P}$ unconditionally satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$ and a black-box satisfying requirement $\mathcal{R}'$ using execution operations X, denoted $\mathcal{P} \models^{\mathcal{M}, \mathcal{R}'}_{X,\infty} \mathcal{R}$, if for any PPT protocol $\mathcal{P}'$ and any model $\mathcal{M}'$, holds:*

$$(\mathcal{P} \cup\!\!\cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\infty} \mathcal{R}' \implies (\mathcal{P} \cup\!\!\cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\infty} \mathcal{R} \quad (3)$$

We say that $\mathcal{P}$ asymptotically satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$ and a black-box satisfying requirement $\mathcal{R}'$ using execution operations X, denoted $\mathcal{P} \models^{\mathcal{M}, \mathcal{R}'}_{X,\textsc{Poly}} \mathcal{R}$, if for any PPT protocol $\mathcal{P}'$ and any model $\mathcal{M}'$, holds:

$$(\mathcal{P} \cup\!\!\cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\textsc{Poly}} \mathcal{R}' \implies (\mathcal{P} \cup\!\!\cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\textsc{Poly}} \mathcal{R} \quad (4)$$

Observation 5 states a property similar to those stated in Observation 4, but for black-box requirement satisfaction. This observation also captures standard weakening between unconditional and asymptotic satisfaction notions. In particular, if a protocol $\mathcal{P}$ unconditionally satisfies requirement $\mathcal{R}$ assuming model $\mathcal{M}$ and a black-box satisfying requirement $\mathcal{R}'$, then $\mathcal{P}$ also asymptotically satisfies $\mathcal{R}$ assuming $\mathcal{M}$ and a black-box satisfying $\mathcal{R}'$.

Observation 5. *Let $\mathcal{P}$ be a PPT protocol, X be execution operations, and $\mathcal{R}$, $\mathcal{M}$, $\mathcal{R}'$ be sets of specifications. Then:*

$$\mathcal{P} \models^{\mathcal{M},\mathcal{R}'}_{X,\infty} \mathcal{R} \Rightarrow \mathcal{P} \models^{\mathcal{M},\mathcal{R}'}_{X,\text{POLY}} \mathcal{R}$$

4.2. The Black-box Composition Lemmas

We now present the *black-box composition lemmas* (Lemmas 6 and 7), which allow composing a protocol which assumes a black-box requirement. Figures 3 and 4 provide visualizations of the two black-box composition lemmas. The lemmas allow multiple layers of composition, as illustrated in Figure 5. The proofs of Lemmas 6 and 7 are in §E.2.

The first lemma we present is the *monolithic black-box composition lemma* (Lemma 6). Notice that Definition 8 (a protocol satisfying a requirement assuming a model and a black-box requirement) depends on a property holding for the composite protocol assuming the combined model, i.e., $(\mathcal{P} \triangledown \mathcal{P}') \models^{(\mathcal{M}\cup\mathcal{M}')}_{X,\text{TYPE}} \mathcal{R}'$. However, we could instead only depend on a property of a monolithic protocol $\mathcal{P}'$ assuming a model $\mathcal{M}'$. We present a visual illustration of the lemma in Figure 3.

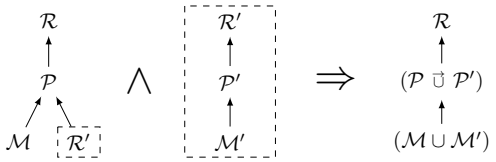

Figure 3. Illustration of Lemma 6. Note that $\mathcal{P}$ and $\mathcal{P}'$ must be non-conflicting PPT protocols.

Lemma 6 (Monolithic black-box composition lemma)**.** *Let $\mathcal{P}$, $\mathcal{P}'$ be non-conflicting PPT protocols, X be execution operations, and $\mathcal{R}$, $\mathcal{M}$, $\mathcal{R}'$, $\mathcal{M}'$ be sets of specifications. Then, for* TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\left[\mathcal{P} \models^{\mathcal{M},\mathcal{R}'}_{X,\text{TYPE}} \mathcal{R}\right] \wedge \left[\mathcal{P}' \models^{\mathcal{M}'}_{X,\text{TYPE}} \mathcal{R}'\right] \implies (\mathcal{P} \triangledown \mathcal{P}') \models^{(\mathcal{M}\cup\mathcal{M}')}_{X,\text{TYPE}} \mathcal{R} \tag{5}$$

The second lemma is the *hybrid black-box composition lemma* (Lemma 7), which is illustrated in Figure 4. This allows a composition where the lower (black-box) protocol itself is a hybrid protocol, i.e., it has a black-box requirement, and consequently, the composite protocol is a hybrid protocol, i.e., has a black-box requirement.

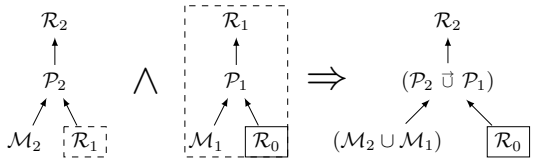

Figure 4. Illustration of Lemma 7. Note that $\mathcal{P}_1$ and $\mathcal{P}_2$ must be non-conflicting PPT protocols.

Lemma 7 (Hybrid black-box composition lemma)**.** *Let $\mathcal{P}_1$, $\mathcal{P}_2$ be non-conflicting PPT protocols, X be execution operations, and $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2, \mathcal{M}_1, \mathcal{M}_2$ be sets of specifications. Then, for* TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\left[\mathcal{P}_2 \models^{\mathcal{M}_2,\mathcal{R}_1}_{X,\text{TYPE}} \mathcal{R}_2\right] \wedge \left[\mathcal{P}_1 \models^{\mathcal{M}_1,\mathcal{R}_0}_{X,\text{TYPE}} \mathcal{R}_1\right] \implies (\mathcal{P}_2 \triangledown \mathcal{P}_1) \models^{(\mathcal{M}_2\cup\mathcal{M}_1),\mathcal{R}_0}_{X,\text{TYPE}} \mathcal{R}_2 \tag{6}$$

Repeated application of Lemma 6 and Lemma 7 allows us to use multiple layers of black boxes. Such hierarchies are common in practice. For example, we can compose three protocols, where two of them are hybrid protocols, and the third one is a monolithic protocol. The composite protocol is a monolithic protocol, i.e., it does not have a black-box requirement. This situation is illustrated in Figure 5.

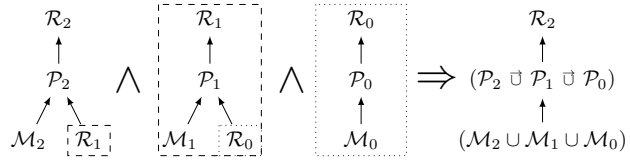

Figure 5. Illustration of applying black-box composition twice. Note that $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2$ must be non-conflicting PPT protocols.

Using Observation 5 from Section 4.1 together with either Lemma 6 or Lemma 7 can be useful for writing proofs. For example, if possible for a particular protocol $\mathcal{P}$, it may be simpler to write a proof that $\mathcal{P}$ unconditionally, rather than asymptotically, satisfies $\mathcal{R}$ assuming a black-box satisfying $\mathcal{R}'$. Then, using Observation 5, it follows that $\mathcal{P}$ also asymptotically satisfies $\mathcal{R}$ assuming a black-box satisfying $\mathcal{R}'$. Next, using Lemma 6 or Lemma 7, $\mathcal{P}$ can be composed with a protocol $\mathcal{P}'$ which asymptotically satisfies $\mathcal{R}'$, and the composite protocol $(\mathcal{P} \triangledown \mathcal{P}')$ would asymptotically satisfy $\mathcal{R}$.

4.3. Interaction Between Instances: the Ask Convention and Model

In §3.1, we defined the composition of protocols $\mathcal{P}$ and $\mathcal{P}'$. The composite protocol $\mathcal{P} \triangledown \mathcal{P}'$ combines the subprotocols of $\mathcal{P}$ and of $\mathcal{P}'$. However, the composition alone does not specify a mechanism for *interaction between the instances of the subprotocols*. This may look unusual: for example, when we consider an "ordinary" program and a function used by that program, then the interaction is defined by the calls to the function from the program.

However, the interaction between instances (typically, of different subprotocols) resembles inter-process communication, rather than a function call. Consider any 'Run' event in which the execution process invokes the algorithm $\mathcal{P}[sp, op_name]$, which we call a (subprotocol) operation, with the state of the instance selected earlier by $\mathcal{A}$, i.e., $sts[ent, sp, ins]$. Such an operation, invoked by the CMoSS execution process on one instance, cannot directly 'call' an operation on another instance ins' of some subprotocol sp', because it does not have access to the state $sts[ent, sp', ins']$ of the other instance.

We present[6] the *ask convention and model*. The *ask convention* is a way in which instances can specify (in their outputs) instances to be invoked, as well as input parameters to those instances. The *ask model* $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}}, Fields}$, presented formally in §F, ensures that the specified instances are invoked as expected in the ask convention. Among other things, the ask model ensures bounded invocation delay, i.e., if at time t the protocol asks to invoke a (new or existing) instance, then the adversary will invoke as asked before time $t + \Delta_{Ask}$. The bound Δ_{Ask} on the time to invoke supports realistic real-time interactions, including the potential context-switching delays associated with invoking another subprotocol (process).

In the ask convention, operation $\mathcal{P}[sp, op_name]$ can *ask* the adversary to invoke a subprotocol operation on another instance at the same entity ent, by specifying the corresponding subprotocol sp', operation op_name' and instance ins', as well as any specific input fields to be provided to the operation. Or, operation $\mathcal{P}[sp, op_name]$ can *ask* the adversary to invoke a *new* instance at the same entity, by specifying a subprotocol sp', an appropriate 'initialization operation' op_name', and input fields.

The $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}}, Fields}$ model ensures that whenever an ask request is carried out, the subprotocol identifier and instance identifier of the *calling instance* are given as part of the input, in the 'calling_sp' and 'calling_ins' input fields, respectively. These two fields should not be used by protocols for other inputs, since they would be overwritten if the protocol is invoked as a result of an ASK request. No other input fields are given to the invoked protocol, except for those specified in the ASK request, with the values outputted by the calling instance, and additionally, the 'calling_sp' and 'calling_ins' input fields, as described above. We refer to the calling instance which initially invokes INIT[7] at the new instance as the *parent instance*, and we refer to the new instance as the *child instance*. As part of the ask convention, when an instance is initialized, i.e., when INIT is first invoked, the instance saves $calling_sp$ and $calling_ins$ as $s.parent_sp$ and $s.parent_ins$.

In the ask convention and model, there is no distinction between invoking and returning. In order to 'return', a child instance simply asks to invoke the parent instance. One subprotocol can have separate operations to handle responses from different subprotocols. The ask convention and model allow such protocol designs. Operations intended to be invoked by specific subprotocol child instances should have names starting with the child subprotocol identifier and a hyphen, e.g. ARR- in the case of CS.ARR-INITRESPSERVER[8] and CS.ARR-RCVRESP. Operations intended to be invoked by parent instances or directly by the adversary should not contain any hyphens, e.g., CS.GETTIME. The ask model uses this naming convention to ensure that operations which are intended to be invoked by specific subprotocol child instances are only invoked by corresponding child instances, e.g., only ARR child instances can invoke CS.ARR-INITRESPSERVER. In addition, a called instance can invoke one of multiple operations at the calling instance. The ask convention and model allow such protocol designs as well. For example, in the protocol $\mathcal{P}_{\text{UnauthCS}}$, there is one initialization response operation for servers (CS.ARR-INITRESPSERVER) and a different one for clients (CS.ARR-INITRESPCLIENT).

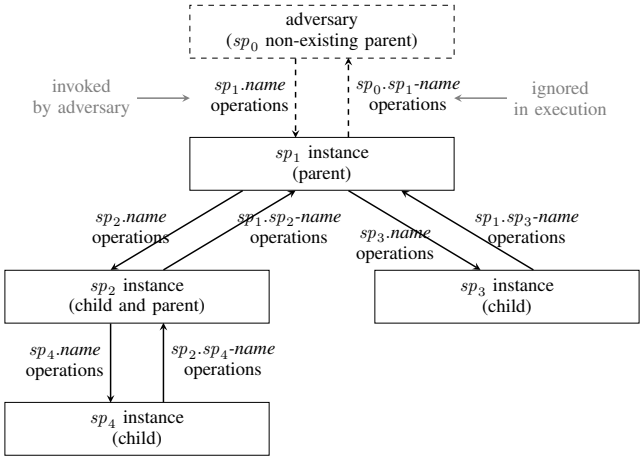

Figure 6. Simplified illustration of the ask convention, enforced by the $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}}, Fields}$ model. Arrows represent outputted asking requests and/or invoking events. Asking requests from children to parents follow the convention of operation names of the form *sp-name*, where *sp* is the child's subprotocol identifier and *name* is any string.

Figure 6 illustrates the ask convention and model. More details are in §F, including an explanation of how the $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}}, Fields}$ model prevents eavesdropping on ASK output fields specified in the $Fields$ parameter.

5. Case Study: Composite Clock Synchronization Protocol

As a case study, we show how we use CMoSS to compose the clock synchronization protocol $\mathcal{P}_{\text{UnauthCS}}$ with the 'lower layer' authenticated request-response protocol $\mathcal{P}_{\text{ARR}}^{\mathcal{S}}$,

6. Other conventions and models can be used, possibly supporting additional features, such the ability to invoke multiple instances from the same event, or the ability to invoke 'unrelated' instances.

7. The ask model does not require any specific operation name for initialization, but in the ask convention, the initialization operation is always called INIT.

8. We use the dot notation to refer to the combination of a subprotocol and a particular operation, e.g., ARR.SENDREQ.

to create a *secure (authenticated) clock synchronization protocol*, denoted $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$, which is illustrated in Figure 7. Secure clock synchronization is a typical network security function, which combines security challenges, such as ignoring spoofed responses, with aspects of reliable distributed computing, such as synchronizing a clock using responses from remote servers, considering unknown network delays and unknown offsets of the remote servers. CMoSS allows modular design of $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$, by defining $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$ as the composition $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}} \equiv \mathcal{P}_{\text{UnauthCS}} \vec{\cup} \mathcal{P}^{\mathcal{S}}_{\text{ARR}}$, where $\mathcal{P}_{\text{UnauthCS}}$ provides clock synchronization assuming an authenticated request-response blackbox, and $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ is an authenticated request-response protocol.

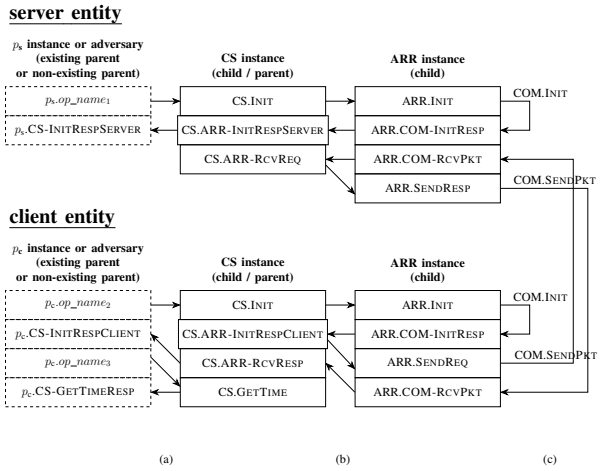

Figure 7. Simplified illustration of composite protocol $\mathcal{P}_{\text{UnauthCS}} \vec{\cup} \mathcal{P}^{\mathcal{S}}_{\text{ARR}}$. Arrows represent outputted asking requests and/or invoking events. (a) Interactions between p_s / p_c parent and CS child instances, where the p_s / p_c instances may be existing parent or non-existing parent, i.e., adversarial, instances. (b) Interactions between CS parent and ARR child instances, which are ensured by the $\mathcal{M}^{\Delta_{\text{Ask}},Fields}_{\text{Ask}}$ model. (c) Communication ensured by the $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ model.

The signature scheme $\mathcal{S}$. The $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ protocol uses a signature scheme $\mathcal{S} = (\text{GEN}, \text{SIGN}, \text{VRFY})$, and assumes it to be secure[9]. Therefore, $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$ also uses $\mathcal{S}$ and assumes it to be secure.

Modular design. The design of $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$ is modular, by designing separately the simpler (unauthenticated) clock synchronization protocol $\mathcal{P}_{\text{UnauthCS}}$ and then combining it with $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$. The design of these two protocols is also modular. Let us describe the design steps for $\mathcal{P}_{\text{UnauthCS}}$ (§H.1). First, we identify the operations of a clock synchronization protocol. Using these, we define the clock synchronization requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$. We present the $\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{\text{CS}}$ model and the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$. By assuming a blackbox for $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$, the design and analysis of the $\mathcal{P}_{\text{UnauthCS}}$ protocol are simpler, compared to directly analyzing the complete (composed) protocol.

Communication between entities. In §G, we present the *COM convention and model*, which support communication between entities and are used in the analysis of the authenticated request-response protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$.

Analysis. We used modular, top-down analysis. In §H.1, we design and analyze the simple unauthenticated clock synchronization protocol $\mathcal{P}_{\text{UnauthCS}}$; we give a proof sketch of why this protocol should satisfy the clock synchronization requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$ assuming the $\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{\text{CS}}$ model and a blackbox protocol satisfying the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$.

In §H.2, we present the $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ protocol, and give a sketch of why it should ensure $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$, assuming (1) the $\mathcal{M}^{\Delta_{com}}_{\text{COM}}[\infty\ ]$ model, and (2) that $\mathcal{S}$ is a secure (i.e., existentially unforgeable) signature scheme. Using Lemma 6, we conclude that $\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}} = \mathcal{P}_{\text{UnauthCS}} \vec{\cup} \mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ should satisfy $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$ assuming (only) models $\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{\text{CS}}$ and $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ [POLY], and that $\mathcal{S}$ is an existentially unforgeable signature scheme. Let us present the relevant theorems. The theorems and analysis do not factor in the $\mathcal{M}^{\text{IO}}_{\Delta_{io}}$ model and the delay between 'Input' and 'Run' events, so they are simplified. Note that we assume the default execution operations mapping X^*, which includes the GET-STATE and SET-STATE X-operations (see §2.2.6).

Theorem 2 ($\mathcal{P}_{\text{UnauthCS}}$ asymptotically satisfies $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$ assuming $\mathcal{M}^{\Delta_{\text{CS}}}_{\text{CS}}$ and a blackbox satisfying $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$)**.** *Let $\Delta_{\text{CS}} \equiv \Delta_{\text{SrvOff}} + 3\Delta_{\text{ARR}} + 5\Delta_{\text{Ask}}$. The (unauthenticated) clock synchronization protocol $\mathcal{P}_{UnauthCS}$ asymptotically satisfies the clock synchronization requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{CS}$ assuming model $\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{CS}$ and a blackbox protocol satisfying requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}$. More precisely:*

$$\mathcal{P}_{UnauthCS} \models^{\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{CS}, \mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}}_{X^*,\text{POLY}} \mathcal{R}^{\Delta_{\text{CS}}}_{CS}.$$

Proof sketch: See §H.1. □

Theorem 3 ($\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ asymptotically satisfies $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$ assuming $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$)**.** *Let $\Delta_{\text{ARR}} \equiv \Delta_{com}$. Then $\mathcal{P}^{\mathcal{S}}_{ARR}$ asymptotically satisfies the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}$, assuming the $\mathcal{M}^{\Delta_{com}}_{COM}[\infty\ ]$ model, and that $\mathcal{S}$ is a secure signature scheme. Namely: for any secure signature scheme $\mathcal{S}$:*

$$\mathcal{P}^{\mathcal{S}}_{ARR} \models^{\mathcal{M}^{\Delta_{com}}_{COM}[\infty]}_{X^*,\text{POLY}} \mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}.$$

Proof sketch: See §H.2. □

Theorem 4 ($\mathcal{P}^{\mathcal{S}}_{\text{AuthCS}}$ asymptotically satisfies $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS}}$ assuming $\mathcal{M}^{\Delta_{\text{CS}}}_{\text{CS}}$ and $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$)**.** *Assume that the signature scheme $\mathcal{S}$ is secure, and let $\Delta_{\text{CS}} = \Delta_{\text{SrvOff}} + 3\Delta_{com} + 5\Delta_{\text{Ask}}$. Then, the composite protocol $\mathcal{P}^{\mathcal{S}}_{AuthCS} \equiv \mathcal{P}_{UnauthCS} \vec{\cup} \mathcal{P}^{\mathcal{S}}_{ARR}$ asymptotically satisfies the clock synchronization requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{CS}$ assuming (only) the $\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{CS}$ and $\mathcal{M}^{\Delta_{com}}_{COM}$ [POLY] models, and that $\mathcal{S}$ is a secure signature scheme. Namely: for any secure signature scheme $\mathcal{S}$:*

$$\mathcal{P}^{\mathcal{S}}_{AuthCS} \models^{\left(\mathcal{M}^{\Delta_{\text{SrvOff}},\Delta_{\text{Ask}}}_{CS} \cup \mathcal{M}^{\Delta_{com}}_{COM}[\text{POLY}]\right)}_{X^*,\text{POLY}} \mathcal{R}^{\Delta_{\text{CS}}}_{CS}.$$

Proof sketch: follows from Theorems 2 and 3 and Lemma 6. □

9. The security required is the standard notion of existential unforgeability; see e.g., Defs. 12.1, 12.2 in [20].

6. Conclusions

We presented CMoSS, a framework that facilitates modular design and analysis of provably secure protocols. CMoSS has support for provably secure compositions of protocols, modular specifications, and real-time concurrency. We are excited about the potential for CMoSS and hope to see it widely adopted as a tool for rigorous, provably secure design and analysis of applied protocols under realistic assumptions.

Acknowledgments

This paper was edited for grammar and editorial changes using Overleaf, Grammarly, and ChatGPT.

References

[1] Backes, M., Pfitzmann, B., Waidner, M.: A general composition theorem for secure reactive systems. In: Theory of Cryptography Conference. pp. 336–354. Springer (2004)

[2] Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology: the journal of the International Association for Cryptologic Research **21**(4), 469–491 (oct 2008), http://www.springerlink.com/openurl.asp?genre=article&issn=0933-2790&volume=21&issue=4&spage=469

[3] Brzuska, C., Cremers, C., Jacobsen, H., Stebila, D., Warinschi, B.: Falsifiability, composability, and comparability of game-based security models for key exchange protocols. Cryptology ePrint Archive, Paper 2024/1215 (2024), https://eprint.iacr.org/2024/1215

[4] Brzuska, C., Delignat-Lavaud, A., Fournet, C., Kohbrok, K., Kohlweiss, M.: State separation for code-based game-playing proofs. In: Peyrin, T., Galbraith, S. (eds.) Advances in Cryptology – ASIACRYPT 2018. pp. 222–249. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_9

[5] Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: iUC: Flexible universal composability made simple. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 191–221. Springer (2019)

[6] 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.

[7] Canetti, R.: Universally composable security. Journal of the ACM (JACM) **67**(5), 1–94 (2020)

[8] Canetti, R., Gennaro, R., Herzberg, A., Naor, D.: Proactive security: Long-term protection against break-ins. RSA Laboratories' CryptoBytes **3**(1), 1–8 (1997)

[9] Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A universally composable treatment of network time. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF). pp. 360–375. IEEE (2017)

[10] Desmedt, Y.: Threshold cryptosystems. In: International Workshop on the Theory and Application of Cryptographic Techniques. pp. 1–14. Springer (1992)

[11] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11), 643–644 (1974)

[12] Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on. pp. 293–302. IEEE (2008)

[13] Goldreich, O., Krawczyk, H.: On the composition of zero-knowledge proof systems. SIAM Journal on Computing **25**(1), 169–192 (1996)

[14] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing (1987)

[15] Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences **28**(2), 270–299 (apr 1984)

[16] Herzberg, A., Leibowitz, H., Syta, E., Wrótniak, S.: MoSS: Modular Security Specifications framework (full version). Cryptology ePrint Archive, Paper 2020/1040 (2020). https://doi.org/10.1007/978-3-030-84252-9_2, https://eprint.iacr.org/2020/1040, https://eprint.iacr.org/2020/1040

[17] Herzberg, A., Leibowitz, H., Syta, E., Wrótniak, S.: MoSS: Modular Security Specifications framework. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 33–63. Springer International Publishing, Cham (2021), full version at: https://eprint.iacr.org/2020/1040

[18] Hofheinz, D., Shoup, V.: Gnuc: A new universal composability framework. Journal of Cryptology **28**(3), 423–508 (2015)

[19] Hofheinz, D., Unruh, D., Müller-Quade, J.: Polynomial runtime and composability. Journal of Cryptology **26**(3), 375–441 (2013). https://doi.org/10.1007/s00145-012-9127-4

[20] Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC Press, second edn. (2015)

[21] 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)

[22] Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology pp. 1–124 (2020)

[23] Liu-Zhang, C.D., Maurer, U.: Synchronous constructive cryptography. In: Theory of Cryptography Conference. pp. 439–472. Springer (2020)

[24] Lochbihler, A., Sefidgar, S.R., Basin, D., Maurer, U.: Formalizing constructive cryptography using crypthol. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). pp. 152–15214. IEEE (2019)

[25] Maurer, U.: Constructive cryptography–a new paradigm for security definitions and proofs. In: Joint Workshop on Theory of Security and Applications. pp. 33–56. Springer (2011)

[26] Wikström, D.: Simplified universal composability framework. In: Theory of Cryptography Conference. pp. 566–595. Springer (2016)

Appendix A. Open Science

All information used in this work is contained within the article and its appendices. There are no additional research artifacts needed to evaluate the research contributions.

Appendix B. Ethical Considerations

This paper does not involve any ethical issues.

Appendix C. CMoSS Execution Process

C.1. Informal Description of the Execution Process

Table 2 lists the notations used in the CMoSS execution process pseudocode (Algorithm 1 in §2.2 and Algorithms 5

Algorithm 5 The ValidVals function

procedure VALIDVALS($T, \tau, vals, EventType, B$)
  if $\tau \leq \max(\{0\} \cup \{e.\tau \mid e \in T\})$ **then return** $\bot$ // *Time must be greater than 0 and greater than the time of the previous event*
  if $EventType \notin \{\text{'Input'}, \text{'Run'}\}$ **then return** $\bot$ // *Event type must be either 'Input' or 'Run'*
  if $EventType = \text{'Input'}$ **then** // *If an 'Input' event*
    // *The operation cannot be an undefined X-operation or an undefined operation of a subprotocol of $\mathcal{P}$*
    if $\left[\begin{array}{l}(vals.Xopr = \top \ \wedge \ X[vals.op_name] = \bot) \\ \vee \left(\begin{array}{l} vals.Xopr \neq \top \ \wedge \ vals.sp \in \mathcal{P}[\cdot] \\ \wedge \ \mathcal{P}[vals.sp, vals.op_name] = \bot \end{array}\right)\end{array}\right]$ **then return** $\bot$
    // *There cannot be more than one value per field specified in inp*
    if $\left[\ \exists (field, v, v') \text{ s.t. } v \neq v' \wedge (field, v), (field, v') \in vals.inp\ \right]$ **then return** $\bot$
    // *There cannot be more than one time value per field specified in pastouts*
    if $\left[\ \exists (field, \tau', \tau'') \text{ s.t. } \tau' \neq \tau'' \wedge (field, \tau'), (field, \tau'') \in vals.pastouts\ \right]$ **then return** $\bot$
    // *There cannot be a time value specified in pastouts such that the time value does not correspond to any previous 'Run' event*
    if $\left[\begin{array}{l}\exists (field', \tau') \in vals.pastouts \text{ s.t.} \\ (\nexists e \in T \text{ s.t. } (e.\tau = \tau' \text{ and } e.EventType = \text{'Run'})\end{array}\right]$ **then return** $\bot$
  else if $EventType = \text{'Run'}$ **then** // *If a 'Run' event*
    // *There must be a previous 'Input' event with time $vals.\tau_{Input}$*
    if $\nexists e \in T$ **s.t.** $\left[\ e.EventType = \text{'Input'} \ \wedge \ e.\tau = vals.\tau_{\text{Input}}\ \right]$ **then return** $\bot$
    // *There cannot be any previous 'Run' event which used the specified input*
    else if $\exists e \in T$ **s.t.** $\left[\begin{array}{l} e.EventType = \text{'Run'} \\ \wedge \ e.\tau_{\text{Input}} = vals.\tau_{\text{Input}}\end{array}\right]$ **then return** $\bot$
  end if

  if $|T| = B$ **then return** $\bot$ // *Too many events*
  if $EventType = \text{'Input'}$ **then** // *If an 'Input' event*
    if $|vals.sp| > B$ **or** $|vals.op_name| > B$ **then return** $\bot$ // *Subprotocol or operation name too large*
    if $vals.op_name = \text{'Bound'}$ **then return** $\bot$ // *Reserved operation name 'Bound'*
    if $\left|\left\{field \;\middle|\; \begin{array}{l}\exists value \text{ s.t. } (field, value) \in vals.inp \\ \text{or } \exists \tau \text{ s.t. } (field, \tau) \in vals.pastouts\end{array}\right\}\right| > B$ **then return** $\bot$ // *Too many unique field names in direct input and past output*
    if $\exists (field, value) \in vals.inp$ **s.t.** $|(field, value)| > B \cdot (|T| + 1)$ **then return** $\bot$ // *A direct input field (name and value) is too large*
  end if

  return $\top$ // *Checks passed*
end procedure

Algorithm 6 The SaveInput function

procedure SAVEINPUT($\tau, T, vals$)
  $(ent, sp, ins, Xopr, op_name, inp, pastouts, vals_{\mathcal{A}}) \leftarrow vals$ // *Extract values from vals*
  $fullinp \leftarrow inp$ // *Copy all fields from direct input to full input*
  $\forall (field, \tau') \in pastouts$: **if** $\nexists value$ **s.t.** $(field, value) \in fullinp$ **then** // *For each past output field, if the field not already in the full input*
    $fullinp \leftarrow fullinp \cup \{(field, e.out[field]) \mid e \in T \text{ s.t. } e.\tau = \tau'\}$ // *Add field with the value from the past output event to full input*
  return NEWEVENT$\left(\begin{array}{c}\tau, \text{'Input'}, ent, sp, ins, Xopr, op_name, \\ vals_{\mathcal{A}}, inp, pastouts, fullinp, \bot, \bot\end{array}\right)$ // *Return new event structure (see Alg. 8)*
end procedure

Algorithm 7 The RUNOPR function

procedure RUNOPR($\tau, s_{\mathcal{A}}, s_X, sts, T, vals, B$)
  $(\tau_{\text{Input}}, vals_{\mathcal{A}}) \leftarrow vals$ // *Get values from* $vals$
  $e_{\text{Input}} \leftarrow e \in T$ **s.t.** $e.\tau = \tau_{\text{Input}}$ // *Get input event with time* τ_{Input}
  $ent \leftarrow e_{\text{Input}}.ent$ // *Get entity from input event*
  $sp \leftarrow e_{\text{Input}}.sp$ // *Get subprotocol id from input event*
  $ins \leftarrow e_{\text{Input}}.ins$ // *Get instance id from input event*
  $Xopr \leftarrow e_{\text{Input}}.Xopr$ // *Get X-operation flag from input event*
  $op_name \leftarrow e_{\text{Input}}.op_name$ // *Get operation name from input event*

  $s_{\mathcal{A}}^{\text{Previous}} \leftarrow s_{\mathcal{A}}$ // *Save* $\mathcal{A}$*'s previous state*
  $sts^{\text{Previous}} \leftarrow sts$ // *Save instances' previous states*

  if $Xopr = \top$ **then**
    $(out, sts, s_X) \leftarrow X[op_name](\tau, vals, sts, s_X, T)$ // *Run* X *operation to get output and new states of all instances*
  else
    $fullinp \leftarrow e_{\text{Input}}.fullinp$ // *Get full input from input event*
    $s \leftarrow sts[ent, sp, ins]$ // *Get current state of relevant instance*
    if $sp \in \mathcal{P}[\cdot]$ **then** // *Determine whether* $\mathcal{P}$ *includes* sp
      $(out, s') \leftarrow \mathcal{P}[sp, op_name](fullinp, s)$ // *Run operation of* $\mathcal{P}$ *to get output and new state for instance*
    else
      $(out, s', s_{\mathcal{A}}) \leftarrow \mathcal{A}[\text{RUN}](sp, op_name, fullinp, s, s_{\mathcal{A}})$ // *Let* $\mathcal{A}$ *select output and new state for instance*
    end if
    $sts[ent, sp, ins] \leftarrow s'$ // *Update instance's state*
  end if
  if $\left[\ \exists(field, v, v')$ **s.t.** $v \neq v' \wedge (field, v), (field, v') \in vals.out\ \right]$ **then** $out \leftarrow \perp$ // *If there is more than one value for some output field, then set the output to* $\perp$
  $\forall ent', sp', ins' \in \{0,1\}^*$: // *For every possible instance, if instance's state size increased too much, then set the state to* $\perp$
    if $|sts[ent', sp', ins']| > |sts^{\text{Previous}}[ent', sp', ins']| + B$ **then** $sts[ent', sp', ins'] \leftarrow \perp$
  if $|s_{\mathcal{A}}| > |s_{\mathcal{A}}^{\text{Previous}}| + B$ **then** $s_{\mathcal{A}} \leftarrow \perp$ // *If* $\mathcal{A}$*'s state size increased too much, then set the state to* $\perp$
  if $|out| > B$ **then** $out \leftarrow \perp$ // *If there are too many elements in the output, then set the output to* $\perp$
  if $\exists(field, value) \in out$ **s.t.** $|(field, value)| > B \cdot (|T| + 1)$ **then** $out \leftarrow \perp$ // *If the size of an output field (name and value) is too large, then set the output to* $\perp$
  return $s_{\mathcal{A}}, s_X, sts,$ NEWEVENT$\left(\begin{array}{c}\tau, \text{'Run'}, ent, sp, ins, Xopr, op_name,\\ vals_{\mathcal{A}}, \perp, \perp, \perp, \tau_{\text{Input}}, out\end{array}\right)$ // *Return states and new event structure (see Alg. 8)*
end procedure

Algorithm 8 The NEWEVENT function

pocedure NEWEVENT$\left(\begin{array}{c}\tau, EventType, ent, sp, ins, Xopr, op_name,\\ vals_{\mathcal{A}}, inp, pastouts, fullinp, \tau_{\text{Input}}, out\end{array}\right)$
  // *Return new event structure with a field containing each input parameter*
  return $\left\{\begin{array}{c}(\text{'}\tau\text{'}, \tau), (\text{'EventType'}, EventType), (\text{'ent'}, ent), (\text{'sp'}, sp), (\text{'ins'}, ins),\\ (\text{'Xopr'}, Xopr), (\text{'op_name'}, op_name), (\text{'vals}_{\mathcal{A}}\text{'}, vals_{\mathcal{A}}),\\ (\text{'inp'}, inp), (\text{'pastouts'}, pastouts), (\text{'fullinp'}, fullinp),\\ (\text{'}\tau_{\text{Input}}\text{'}, \tau_{\text{Input}}), (\text{'out'}, out)\end{array}\right\}$
end procedure

TABLE 2. LIST OF NOTATIONS USED IN THE CMOSS EXECUTION PROCESS (ALGORITHMS 1 AND 5 TO 8).

General Variables and Notation	
Variable Name / Notation	**Description**
$\mathcal{P}$	Protocol
$\mathcal{A}$	Adversary
X	Set of operations that extend the execution process
1^κ	Security parameter input to the execution process
T	Execution transcript
$\mathcal{P}[\cdot]$	The set of subprotocols of $\mathcal{P}$
State Variables	
Variable Name	**Description**
s_X	State of execution operations X
$s_{\mathcal{A}}$	State of adversary $\mathcal{A}$
$sts[ent, sp, ins]$	State of instance (ent, sp, ins)
Bound-related Variables	
Variable Name	**Description**
B	Bound used for number of input and output fields, allowed increase in field sizes, allowed increase in state sizes, and allowed number of events
$s_{\mathcal{A}}^{\text{Previous}}$	Adversary's state at the beginning of a 'Run' event
sts^{Previous}	Instances' states at the beginning of a 'Run' event
Event Variables and Event Field Names	
Variable Name ([Event Field Name])	**Description**
τ ['τ']	Time of current event
$EventType$ ['EventType']	Type of event: 'Input' or 'Run'
$vals$	Values provided by the adversary for this event
ent ['ent']	Entity identifier of the instance
sp ['sp']	Subprotocol identifier of the instance
ins ['ins']	Instance identifier of the instance
$Xopr$ ['Xopr']	Flag indicating whether to run an X operation in a 'Run' event
op_name ['op_name']	Operation name of the operation to invoke
$vals_{\mathcal{A}}$ ['vals$_{\mathcal{A}}$']	Adversary values (not given as input to protocol instances)
inp ['inp']	Input value(s)
pastouts ['pastouts']	Set of pairs of output field names and previous 'Run' event times, used to specify which outputs from past events should be added to the input
$fullinp$ ['fullinp']	Full input, including the inp values and possibly values from previous outputs, as specified *pastouts*
τ_{Input} ['τ_{Input}']	Time of past 'Input' event to be used in current 'Run' event
e_{Input}	Past 'Input' event to be used in current 'Run' event
s	State of instance
s'	Updated state of instance
out ['out']	Operation's output
$newEvent$	New event structure, to be added the execution transcript

to 8). The main algorithm of the CMoSS execution process (Algorithm 1 in §2.2) uses the functions shown in Algorithms 5 to 8. We now informally describe the structure of the CMoSS execution process.

The execution process (Algorithm 1 in §2.2) begins by initializing the state of the adversary and of every possible instance. We then enter the *event loop* (Lines 5-14), in which each iteration corresponds to a single *event* dictated by the adversary. That is, the adversary decides which operation (op_name) of the protocol to invoke, at which entity (ent), with which inputs (inp), and at what time. The execution process follows the adversary's directive and either saves input (in an 'Input' event) or runs an operation (in a 'Run' event) accordingly (Lines 8-12). In each iteration, an event structure is created, containing event fields and values, and is added to the execution transcript T (Line 13). Later, predicates refer to these event entries to evaluate models and requirements over complete executions. Upon termination, the execution process outputs the transcript T of the events that occurred during the execution (Line 7). The events in the transcript do not include indices, but they are well-ordered by their times. As a convention, we use $T[i]$ to denote the ith event in the execution transcript for $i \in \{1, \ldots |T|\}$, and $\perp$ for $i \notin \{1, \ldots |T|\}$.

TABLE 3. LIST OF FIELDS AND THEIR VALUES ADDED TO EVENTS IN 'INPUT' AND 'RUN' EVENTS IN THE CMOSS EXECUTION PROCESS (IN ALGORITHMS 6 TO 8). (*) VALUE COPIED FROM THE 'INPUT' EVENT USED IN THE 'RUN' EVENT.

Event Fields and Values			
Event Field Name	**Value in 'Input' Events**	**Value in 'Run' Events**	**Description**
'τ'	τ	τ	Time of current event
'EventType'	'Input'	'Run'	Type of event: 'Input' or 'Run'
'ent'	ent	ent (*)	Entity identifier of the instance
'sp'	sp	sp (*)	Subprotocol identifier of the instance
'ins'	ins	ins (*)	Instance identifier of the instance
'Xopr'	$Xopr$	$Xopr$ (*)	Flag indicating whether to run an X operation in a 'Run' event
'op_name'	op_name	op_name (*)	Operation name of the operation to invoke
'vals$_{\mathcal{A}}$'	$vals_{\mathcal{A}}$	$vals_{\mathcal{A}}$	Adversary values (not given as input to protocol instances)
'inp'	inp	$\perp$	Input value(s)
'pastouts'	*pastouts*	$\perp$	Set of pairs of output field names and previous 'Run' event times, used to specify which outputs from past events should be added to the input
'fullinp'	$fullinp$	$\perp$	Full input, including the inp values and possibly values from previous outputs, as specified *pastouts*
'τ_{Input}'	$\perp$	τ_{Input}	Time of past 'Input' event to be used in current 'Run' event
'out'	$\perp$	out	Operation's output

C.2. Rules for Event Values

The execution process checks the adversary-provided values in each iteration using the ValidVals function (Algorithm 5).

Execution ends if the following rules are broken in any iteration:

- The time of events must be positive and strictly increasing.
- The event type must be either 'Input' or 'Run'.
- To ensure bounded runtime (see the comment in the caption of Algorithm 1), the number of events in T cannot be too large.

In 'Input' events, execution ends if the following rules are broken:

- The operation cannot be an undefined operation of a sp in $\mathcal{P}[\cdot]$ or an undefined X-operation.
- The direct input fields must be unique, i.e., there cannot be two fields in inp with the same field name but different values.
- The fields in *pastouts* must have non-ambiguous times of earlier 'Run' events.
- To ensure bounded runtime, the sizes of sp and op_name cannot be too large, op_name cannot be the reserved name 'Bound', the number of unique field names in direct input and past output cannot be too large, and the sizes of direct input field names and values, cannot be too large.

In 'Run' events, execution ends if the following rule is broken:

- 'τ_{Input}' must be the time of an earlier, unused 'Input' event.

In addition, in 'Run' events, the execution process checks the output and states in the RunOpr function (Algorithm 7). If the following rules are broken, then the execution process sets the output or state(s) to $\perp$, but the event completes and execution continues.

- The output fields must be unique, i.e., there cannot be two fields in out with the same field name but different values. If there are, out is set to $\perp$.
- To ensure bounded runtime, the sizes of instances' states and the adversary's state cannot increase too much (since the beginning of the event). Any state whose size increases too much is set to $\perp$. Also to ensure bounded runtime, the sizes of output field names and values cannot be too large. If any output field (name and value) is too large, then out is set to $\perp$.

C.3. Event Fields

In each iteration of an execution, an event structure containing fields relevant to that iteration is added to the execution transcript. The fields and values added to each event depend on the event type ('Input' or 'Run'). The fields, their use, and their values in 'Input' and 'Run' events are listed in Table 3.

As shown in Table 3, the values of some fields are set to $\perp$. In 'Input' events, these are the 'τ_{Input}' and 'out' fields, because these fields are not used in 'Input' events. In 'Run' events, these are the 'inp', 'pastouts', and 'fullinp' fields, because the values of these fields are already recorded in the corresponding 'Input' event.

Table 3 also shows that some field values in 'Run' events are copied from the 'Input' event used in that 'Run' event so that they match in the two corresponding events; specifically, these are the values of the 'ent', 'sp', 'ins', 'Xopr', and 'op_name' fields. These values are not set to $\perp$ for convenience when later writing model and requirement predicates, so that the entity, subprotocol, instance, and operation can be easily identified directly from both 'Input' and 'Run' events.

If the adversary provides other values (using $\mathcal{A}$[SELECT]) for the 'τ_{Input}' or 'out' fields in 'Input' events, or for the 'ent', 'sp', 'ins', 'Xopr', 'op_name', 'inp', 'pastouts', or 'fullinp' fields in 'Run' events, then the adversary's values are ignored.

One field which we have not discussed is the '$\text{vals}_{\mathcal{A}}$' field. The adversary chooses the value $vals_{\mathcal{A}}$ for this field in both 'Input' and 'Run' events. As noted in Table 3, $vals_{\mathcal{A}}$ is not given as input to the protocol instances. It is also not given as input to adversarial instances (the adversary should be able to know it anyway, since it chose it), but it is given as part of the input to X-operations (in addition to other information, including the current execution transcript). The purpose of $vals_{\mathcal{A}}$ is essentially meta-information. The adversary can specify additional information in $vals_{\mathcal{A}}$, e.g., to identify the current event as a special event which shows that the adversary 'wins' in some requirement game. This can later be convenient for writing model and requirement predicates.

Appendix D.
Entity Corruptions

In this section, we discuss entity corruptions, which were briefly considered in §2.2.6. We begin by formally defining two X-operations useful for modeling entity corruptions. Then, we define two corruption models: a threshold fault model and a self-stabilization model.

As discussed in §2.2.6, two useful X-operations for entity faults allow letting the adversary see and change the states of an entity's instances. In Algorithm 9, we present pseudocode for the GET-STATE and SET-STATE X-operations[10] for entity faults, which is based on the informal descriptions in MoSS [17].

To make it more convenient to identify entities corrupted in executions, we define the EntXOpr, Corrupt, and Benign functions in Algorithm 10. The function $\mathsf{EntXOpr}(T, Opr)$ returns the set of all entities at which any of the X-operations specified in the set Opr was run. The parameter Opr allows choosing different corruption or other X-operations. The functions Corrupt and Benign are defined with the corruption X-operations fixed to 'Set-state' and 'Get-state'. The function $\mathsf{Corrupt}(T)$ returns the set of all entities at which either the 'Set-state' or the 'Get-state' X-operation was run, while the function $\mathsf{Benign}(T)$ returns the set of all entities at which neither the 'Set-state' nor the 'Get-state' X-operation was run.

Protocols may not use the identifiers ent, sp, ins used in the execution process (in fact, protocol instances may not know their execution process identifiers, unless the adversary gives instances their identifiers). Protocols may use protocol-specific *addresses* to identify and communicate with different protocol instances, which allows more flexibility, such as having multiple addresses per instance. The CorruptAddr function defined in Algorithm 10 allows identifying 'corrupt addresses' assigned by a particular subprotocol sp, i.e., addresses assigned by subprotocol sp to instances at corrupt entities. The function $\mathsf{CorruptAddr}(T, sp, addr)$ returns $\top$ if address $addr$ was assigned, using operation name sp_0'-InitResp', to an instance at a corrupt entity, and returns $\bot$ otherwise.

Using GET-STATE and SET-STATE, we can model different types of corruptions and faults. For example, we can define a fault model in which the adversary has access to the state of all instances at one entity. We can also define a threshold faults model, which limits the numbers of corrupted entities with specific subprotocols and roles to some functions of the total number of such entities. We define the threshold faults model predicate $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi$ (Algorithm 11), which ensures that different corruption thresholds are kept for different types of entities. Specifically, the $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi$ model predicate checks that for each tuple $(sp, role, Opr, f)$ in the parameter $Roles$, the function f bounds the fraction of the relevant entities which were corrupted using any of the X-operations in Opr, where the relevant entities include all entities which had at least one instance of subprotocol sp and were initialized with role $role$.

The $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi$ predicate uses the functions EntSpRole and EntXOpr. The function $\mathsf{EntSpRole}(T, sp, role)$, defined in Algorithm 11, returns the set of all entities which had at least one instance of subprotocol sp initialized with role $role$. The function EntXOpr is described above and defined in Algorithm 10.

The $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi$ threshold faults model predicate can be compared to the $\mathcal{M}^{\mathsf{F}}$ and $\mathcal{M}^{|\mathsf{F}|\leq f}$ models defined in MoSS [16]. The first difference is that the MoSS model $\mathcal{M}^{|\mathsf{F}|\leq f}$ does not have a parameter to specify corrupting X-operations - they are fixed to be the GET-STATE, SET-OUTPUT, and SET-STATE X-operations described in [16]. The second difference is that the MoSS model $\mathcal{M}^{|\mathsf{F}|\leq f}$ does not enforce the threshold on different sets of entities, but rather on all entities in the execution.

Next, we define the self-stabilization model predicate $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi$ (Algorithm 12). The predicate $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi$ allows the adversary to corrupt entities' states using the SET-STATE X operation as long as it is sufficiently early in the execution (specifically, before the last Δ time of the execution), and the entities had at least one instance of subprotocol sp initialized with role $role$, for some $(sp, role) \in R$. The $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi$ model predicate does not allow the adversary to corrupt other entities' states using the SET-STATE X-operation, and it does not allow the adversary to view any entity's states using the GET-STATE X operation. A similar variant of the self-stabilization model predicate $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi$ could be defined which does not prevent GET-STATE corruptions.

10. In Algorithm 9, we use the keyword 'X-operation' instead of 'procedure' to make it clear that these are X-operations.

Appendix E.
Proofs

E.1. Proofs of the Composition Properties

We present proofs of the four composition properties from §3.2.

Lemma 1 (Composition is associative)**.** *For any three protocols $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2$: $\mathcal{P}_2 \vec{\cup} (\mathcal{P}_1 \vec{\cup} \mathcal{P}_0) = (\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \vec{\cup} \mathcal{P}_0$.*

Proof. Using Definition 6, we will show that for any $sp, op_name \in \{0,1\}^*$:

$$(\mathcal{P}_2 \vec{\cup} (\mathcal{P}_1 \vec{\cup} \mathcal{P}_0))[sp, op_name] = ((\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \vec{\cup} \mathcal{P}_0)[sp, op_name]$$

Let $sp, op_name \in \{0,1\}^*$. There are three cases: only $\mathcal{P}_0$ includes sp; $\mathcal{P}_1$ includes sp but $\mathcal{P}_2$ does not include sp; $\mathcal{P}_2$ includes sp.

Case 1: If only $\mathcal{P}_0$ includes sp, then using Definition 6, we get the following.

$$(\mathcal{P}_2 \vec{\cup} \mathcal{P}_1)[sp, op_name] = \mathcal{P}_1[sp, op_name] = \bot$$
$$((\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \vec{\cup} \mathcal{P}_0)[sp, op_name] = \mathcal{P}_0[sp, op_name]$$
$$(\mathcal{P}_1 \vec{\cup} \mathcal{P}_0)[sp, op_name] = \mathcal{P}_0[sp, op_name]$$
$$(\mathcal{P}_2 \vec{\cup} (\mathcal{P}_1 \vec{\cup} \mathcal{P}_0))[sp, op_name] = \mathcal{P}_0[sp, op_name]$$

Algorithm 9 The GET-STATE and SET-STATE X-operations

X-operation GET-STATE($\tau, vals, sts, s_X, T$)
  $(\tau_{\text{Input}}, vals_{\mathcal{A}}) \leftarrow vals$ // *Get values from* $vals$
  $e_{\text{Input}} \leftarrow e \in T$ **s.t.** $e.\tau = \tau_{\text{Input}}$ // *Get the event with time* τ_{Input} *from the transcript*
  $stsAtEnt \leftarrow \emptyset$
  $\forall sp, ins \in \{0,1\}^*$ **s.t.** $sts[e_{\text{Input}}.ent, sp, ins] \neq \{(\text{`}1^\kappa\text{'}, 1^\kappa)\}$: // *Find states of all instances at entity* $e_{Input}.ent$ *which have been modified since initialization*
    $stsAtEnt \leftarrow stsAtEnt \cup (sp, ins, sts[e_{\text{Input}}.ent, sp, ins])$ // *Add these states to the set* $stsAtEnt$
  return $((\text{`stsAtEnt'}, stsAtEnt), sts, s_X)$ // *Output all modified states of instances at entity* ent
end X-operation

X-operation SET-STATE($\tau, vals, sts, s_X, T$)
  $(\tau_{\text{Input}}, vals_{\mathcal{A}}) \leftarrow vals$ // *Get values from* $vals$
  $e_{\text{Input}} \leftarrow e \in T$ **s.t.** $e.\tau = \tau_{\text{Input}}$ // *Get the event with time* τ_{Input} *from the transcript*
  $\forall sp, ins \in \{0,1\}^*$ **s.t.** $e_{\text{Input}}.inp[\text{`sts'}][sp, ins] \neq \perp$: // *Overwrite the states at entity* $e_{Input}.ent$ *according to the values in* $e_{Input}.inp$
    $sts[e_{\text{Input}}.ent, sp, ins] \leftarrow e_{\text{Input}}.inp[\text{`sts'}][sp, ins]$
  return $(\perp, sts, s_X)$ // *Output* $\perp$*, but return the updated states* sts
end X-operation

Thus, in this case,

$$(\mathcal{P}_2 \,\overline{\cup}\, (\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0))[sp, op_name] = ((\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1) \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name]$$

Case 2: If $\mathcal{P}_1$ includes sp but $\mathcal{P}_2$ does not include sp, then using Definition 6, we get the following.

$$(\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1)[sp, op_name] = \mathcal{P}_1[sp, op_name]$$
$$((\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1) \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name] = \mathcal{P}_1[sp, op_name]$$
$$(\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name] = \mathcal{P}_1[sp, op_name]$$
$$(\mathcal{P}_2 \,\overline{\cup}\, (\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0))[sp, op_name] = \mathcal{P}_1[sp, op_name]$$

Thus, in this case,

$$(\mathcal{P}_2 \,\overline{\cup}\, (\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0))[sp, op_name] = ((\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1) \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name]$$

Case 3: If $\mathcal{P}_2$ includes sp, then using Definition 6, we get the following.

$$(\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1)[sp, op_name] = \mathcal{P}_2[sp, op_name]$$
$$((\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1) \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name] = \mathcal{P}_2[sp, op_name]$$
$$(\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name] = \begin{cases} \mathcal{P}_1[sp, op_name] \\ \textbf{or } \mathcal{P}_0[sp, op_name] \end{cases}$$
$$(\mathcal{P}_2 \,\overline{\cup}\, (\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0))[sp, op_name] = \mathcal{P}_2[sp, op_name]$$

Thus, in this case,

$$(\mathcal{P}_2 \,\overline{\cup}\, (\mathcal{P}_1 \,\overline{\cup}\, \mathcal{P}_0))[sp, op_name] = ((\mathcal{P}_2 \,\overline{\cup}\, \mathcal{P}_1) \,\overline{\cup}\, \mathcal{P}_0)[sp, op_name]$$

□

Lemma 2 (Composition is not commutative)**.** *There exist two protocols* $\mathcal{P}, \mathcal{P}'$ *such that:* $\mathcal{P} \,\overline{\cup}\, \mathcal{P}' \neq \mathcal{P}' \,\overline{\cup}\, \mathcal{P}$.

Proof. Let $\mathcal{F}$ and $\mathcal{G}$ be two different algorithms, i.e., $\mathcal{F} \neq \mathcal{G}$, such that $\mathcal{F} \neq \perp$, $\mathcal{G} \neq \perp$. Let $\mathcal{P}_F$ be a protocol, i.e., a mapping from pairs of strings to algorithms or to $\perp$, such that $\mathcal{P}_F[\text{`A'}, \text{`foo'}] = \mathcal{F}$, and let $\mathcal{P}_G$ be a protocol, i.e., a mapping from pairs of strings to algorithms or to $\perp$, such that $\mathcal{P}_G[\text{`A'}, \text{`foo'}] = \mathcal{G}$. Notice that this implies that interface 'A' is included in both $\mathcal{P}_F$ and $\mathcal{P}_G$. Following Definition 6, all operations with subprotocols which are included in $\mathcal{P}_F$ are defined in $\mathcal{P}_F \,\overline{\cup}\, \mathcal{P}_G$ as in $\mathcal{P}_F$, and all operations with subprotocols which are included in $\mathcal{P}_G$ are defined in $\mathcal{P}_G \,\overline{\cup}\, \mathcal{P}_F$ as in $\mathcal{P}_G$. This implies that $(\mathcal{P}_F \,\overline{\cup}\, \mathcal{P}_G)[\text{`A'}, \text{`foo'}] = \mathcal{P}_F[\text{`A'}, \text{`foo'}] = \mathcal{F}$ and $(\mathcal{P}_G \,\overline{\cup}\, \mathcal{P}_F)[\text{`A'}, \text{`foo'}] = \mathcal{P}_G[\text{`A'}, \text{`foo'}] = \mathcal{G}$. Since $\mathcal{F}$ and $\mathcal{G}$ are algorithms such that $\mathcal{F} \neq \mathcal{G}$, this implies that: $(\mathcal{P}_F \,\overline{\cup}\, \mathcal{P}_G)[\text{`A'}, \text{`foo'}] \neq (\mathcal{P}_G \,\overline{\cup}\, \mathcal{P}_F)[\text{`A'}, \text{`foo'}]$. That is, the pair of strings ('A', 'foo') is mapped to a different algorithm in $(\mathcal{P}_F \,\overline{\cup}\, \mathcal{P}_G)$ than in $(\mathcal{P}_G \,\overline{\cup}\, \mathcal{P}_F)$. Namely, $(\mathcal{P}_F \,\overline{\cup}\, \mathcal{P}_G) \neq (\mathcal{P}_G \,\overline{\cup}\, \mathcal{P}_F)$, i.e., composition is not commutative. □

Lemma 3 (Composition of non-conflicting protocols is commutative)**.** *For any non-conflicting protocols* $\mathcal{P}$ *and* $\mathcal{P}'$ *holds:* $\mathcal{P} \,\overline{\cup}\, \mathcal{P}' = \mathcal{P}' \,\overline{\cup}\, \mathcal{P}$.

Proof. Let $\mathcal{P}$ and $\mathcal{P}'$ be non-conflicting protocols. Using Definition 6, we will show that for any $sp, op_name \in \{0,1\}^*$:

$$(\mathcal{P} \,\overline{\cup}\, \mathcal{P}')[sp, op_name] = (\mathcal{P}' \,\overline{\cup}\, \mathcal{P})[sp, op_name]$$

Let $sp, op_name \in \{0,1\}^*$. There are four cases: both $\mathcal{P}$ and $\mathcal{P}'$ include sp; only $\mathcal{P}$ includes sp; only $\mathcal{P}'$ includes sp; neither $\mathcal{P}$ nor $\mathcal{P}'$ include sp.

Case 1: If both $\mathcal{P}$ and $\mathcal{P}'$ include sp, then using Definition 6, we get the following.

$$(\mathcal{P} \,\overline{\cup}\, \mathcal{P}')[sp, op_name] = \mathcal{P}[sp, op_name]$$
$$(\mathcal{P}' \,\overline{\cup}\, \mathcal{P})[sp, op_name] = \mathcal{P}'[sp, op_name]$$

Since $\mathcal{P}$ and $\mathcal{P}'$ both include sp, then $\exists op_name', op_name'' \in \{0,1\}^*$ such that

Algorithm 10 The EntXOpr, Corrupt, Benign, and CorruptAddr functions

// Return the set of entities corrupted using any of the X-operations in the set Opr

1: **procedure** EntXOpr(T, Opr)

2: **return** $\left\{ e.ent \;\middle|\; \begin{array}{l} e \in T \\ \wedge\ e.EventType = \text{`Run'} \\ \wedge\ e.Xopr = \top \\ \wedge\ e.op_name \in Opr \end{array} \right\}$

3: **end procedure**

// Return the set of entities corrupted using the SET-STATE *or* GET-STATE *X-operation*

4: **procedure** CORRUPT(T)

5: **return** EntXOpr(T, {'Set-state', 'Get-state'})

6: **end procedure**

// Return the set of entities not corrupted using the SET-STATE *or* GET-STATE *X-operation*

7: **procedure** BENIGN(T)

8: **return** $\{e.ent \mid e \in T\} \setminus$ Corrupt(T)

9: **end procedure**

// Return $\top$ if the address was assigned, using operation name sp'-InitResp', to an instance at a corrupt entity

10: **procedure** CORRUPTADDR(T, sp, $addr$)

11: **return** $\top$ **iff** $\exists e \in T$ **s.t.**

$$\left[\begin{array}{l} e.EventType = \text{`Input'} \\ \wedge\ e.Xopr \neq \top \\ \wedge\ e.op_name = sp\text{`-InitResp'} \\ \wedge\ e.inp.addr = addr \\ \wedge\ e.ent \in \text{Corrupt}(T) \end{array} \right]$$

12: **end procedure**

$\mathcal{P}[sp, op_name'] \neq \perp$ and $\mathcal{P}'[sp, op_name''] \neq \perp$. Suppose for contradiction that $\mathcal{P}[sp, op_name] \neq \mathcal{P}'[sp, op_name]$. Then, using Definition 7, it follows that $\mathcal{P}$, $\mathcal{P}'$ have a conflict, i.e., are not non-conflicting. This is a contradiction, because we assumed that $\mathcal{P}$, $\mathcal{P}'$ are non-conflicting. Thus, $\mathcal{P}[sp, op_name]$ must equal $\mathcal{P}'[sp, op_name]$, so $\mathcal{P}[sp, op_name] = \mathcal{P}'[sp, op_name]$. This implies that in this case, $(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = (\mathcal{P}' \cup \mathcal{P})[sp, op_name]$.

Case 2: If only $\mathcal{P}$ includes sp, then using Definition 6, we get the following.

$$(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = \mathcal{P}[sp, op_name]$$
$$(\mathcal{P}' \cup \mathcal{P})[sp, op_name] = \mathcal{P}[sp, op_name]$$

Thus, in this case, $(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = (\mathcal{P}' \cup \mathcal{P})[sp, op_name]$.

Algorithm 11 Threshold faults model predicate $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi$

// For each $(sp, role, Opr, f)$ tuple in $Roles$, consider the set of entities which had at least one instance of subprotocol sp and were initialized with role $role$. The fraction of such entities which were corrupted using any of the X-operations in Opr does not exceed the threshold specified by the function f.

1: **Allowed failure probability** $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\beta \equiv 0$

2: **Subprotocols** $\mathcal{M}^{\text{ThXOpr}}_{Roles}.Sp \equiv \emptyset$

3: **Model predicate** $\mathcal{M}^{\text{ThXOpr}}_{Roles}.\pi(T)$

4: **return** $\top$ **iff**:

5: $\forall (sp, role, Opr, f) \in Roles$:

6: $$\left| \begin{array}{l} \text{EntSpRole}(T, sp, role) \\ \cap\ \text{EntXOpr}(T, Opr) \end{array} \right| \leq f\left(|\ \text{EntSpRole}(T, sp, role)|\right)$$

7: **end model predicate**

// Return the set of entities which had at least one instance of subprotocol sp initialized with role $role$

8: **procedure** EntSpRole(T, sp, $role$)

9: **return** $\left\{ e.ent \;\middle|\; \begin{array}{l} e \in T \\ \wedge\ e.EventType = \text{`Input'} \\ \wedge\ e.sp = sp \\ \wedge\ e.op_name = \text{`Init'} \\ \wedge\ e.inp.role = role \end{array} \right\}$

10: **end procedure**

Case 3: If only $\mathcal{P}'$ includes sp, then using Definition 6, we get the following.

$$(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = \mathcal{P}'[sp, op_name]$$
$$(\mathcal{P}' \cup \mathcal{P})[sp, op_name] = \mathcal{P}'[sp, op_name]$$

Thus, in this case, $(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = (\mathcal{P}' \cup \mathcal{P})[sp, op_name]$.

Case 4: If neither $\mathcal{P}$ nor $\mathcal{P}'$ includes sp, then using Definition 6, we get the following.

$$(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = \mathcal{P}'[sp, op_name]$$
$$(\mathcal{P}' \cup \mathcal{P})[sp, op_name] = \mathcal{P}[sp, op_name]$$

Since neither $\mathcal{P}$ nor $\mathcal{P}'$ includes sp, then $\mathcal{P}'[sp, op_name] = \perp$ and $\mathcal{P}[sp, op_name] = \perp$. This implies that in this case,

$$(\mathcal{P} \cup \mathcal{P}')[sp, op_name] = (\mathcal{P}' \cup \mathcal{P})[sp, op_name]$$

□

Lemma 4 (Composition preserves the non-conflicting property)**.** *Let $\{\mathcal{P}_i\}_{i=0}^{2}$ be a set of protocols such that $\mathcal{P}_0, \mathcal{P}_1$ are non-conflicting and $\mathcal{P}_0, \mathcal{P}_2$ are non-conflicting, and let $\mathcal{P}_{12} \equiv \mathcal{P}_1 \cup \mathcal{P}_2$. Then $\mathcal{P}_0, \mathcal{P}_{12}$ are also non-conflicting.*

Proof. For contradiction, suppose that $\mathcal{P}_2 \cup \mathcal{P}_1$, $\mathcal{P}_0$ are

Algorithm 12 Self-stabilization model predicate $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi$

// Entities which had at least one instance of subprotocol sp initialized with role role, for some $(sp, role) \in R$ can be corrupted using the SET-STATE *X-operation before the last Δ time of the execution, but they cannot be corrupted using* SET-STATE *later. Other entities cannot be corrupted using* SET-STATE *at all. No entity can be corrupted using the* GET-STATE *X-operation.*

Allowed failure probability $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\beta \equiv 0$

Subprotocols $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.Sp \equiv \emptyset$

Model predicate $\mathcal{M}^{\text{Self-stabil}}_{\Delta,R}.\pi(T)$

// Find last event in execution

$e_{\text{Last}} \leftarrow$ event in T **s.t.** $e_{\text{Last}}.\tau = \max\{e'.\tau \mid e' \in T\}$

return $\top$ **iff**:

// The SET-STATE *X-operation was not run in the last Δ time of the execution*

$$\nexists e \in T \textbf{ s.t. } \left[\begin{array}{l} e.EventType = \text{`Run'} \\ \wedge\ e.Xopr = \top \\ \wedge\ e.op_name = \text{`Set-state'} \\ \wedge\ e_{\text{Last}}.\tau - e.\tau \leq \Delta \end{array}\right]$$

// The SET-STATE *X-operation was not run at any entity which did not have any instances of subprotocol sp initialized with role role (see Alg. 11) for some $(sp, role) \in R$*

$$\wedge\ \nexists e \in T \textbf{ s.t. } \left[\begin{array}{l} e.EventType = \text{`Run'} \\ \wedge\ e.Xopr = \top \\ \wedge\ e.op_name = \text{`Set-state'} \\ \wedge\ \left[\begin{array}{l} \nexists (sp, role) \in R \textbf{ s.t.} \\ e.ent \in \mathsf{EntSpRole}(T, sp, role) \end{array}\right] \end{array}\right]$$

// The GET-STATE *X-operation was not run in the execution*

$$\wedge\ \nexists e \in T \textbf{ s.t. } \left[\begin{array}{l} e.EventType = \text{`Run'} \\ \wedge\ e.Xopr = \top \\ \wedge\ e.op_name = \text{`Get-state'} \end{array}\right]$$

end model predicate

not non-conflicting. Using Definition 7, this implies that for some $sp \in \{0,1\}^*$, one of the following two cases is true:

Case 1: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name] \neq \perp\ \wedge \mathcal{P}_0[sp, op_name'] \neq \perp \wedge \\ (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name] \neq \mathcal{P}_0[sp, op_name] \end{array}\right)$$

Case 2: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} \mathcal{P}_0[sp, op_name] \neq \perp\ \wedge (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name'] \neq \perp \wedge \\ \mathcal{P}_0[sp, op_name] \neq (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name] \end{array}\right)$$

Using Definition 6, for any $sp \in \{0,1\}^*$, either $\forall op_name \in \{0,1\}^* : (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name] = \mathcal{P}_2[sp, op_name]$ or $\forall op_name \in \{0,1\}^* : (\mathcal{P}_2 \cup \mathcal{P}_1)[sp, op_name] = \mathcal{P}_1[sp, op_name]$. This implies that one of the following four cases is true:

Case 1a: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} \mathcal{P}_2[sp, op_name] \neq \perp\ \wedge \mathcal{P}_0[sp, op_name'] \neq \perp \wedge \\ \mathcal{P}_2[sp, op_name] \neq \mathcal{P}_0[sp, op_name] \end{array}\right)$$

Case 1b: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} \mathcal{P}_1[sp, op_name] \neq \perp\ \wedge \mathcal{P}_0[sp, op_name'] \neq \perp \wedge \\ \mathcal{P}_1[sp, op_name] \neq \mathcal{P}_0[sp, op_name] \end{array}\right)$$

Case 2a: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} \mathcal{P}_0[sp, op_name] \neq \perp\ \wedge \mathcal{P}_2[sp, op_name'] \neq \perp \wedge \\ \mathcal{P}_0[sp, op_name] \neq \mathcal{P}_2[sp, op_name] \end{array}\right)$$

Case 2b: $\exists op_name, op_name' \in \{0,1\}^*$ s.t.
$$\left(\begin{array}{l} \mathcal{P}_0[sp, op_name] \neq \perp\ \wedge \mathcal{P}_1[sp, op_name'] \neq \perp \wedge \\ \mathcal{P}_0[sp, op_name] \neq \mathcal{P}_1[sp, op_name] \end{array}\right)$$

Cases 1a and 2a imply that $\mathcal{P}_2$, $\mathcal{P}_0$ are not non-conflicting, which is a contradiction. Cases 1b and 2b imply that $\mathcal{P}_1$, $\mathcal{P}_0$ are not non-conflicting, which is a contradiction. Thus, all four of the cases lead to contradictions. This implies that $\mathcal{P}_2 \cup \mathcal{P}_1$, $\mathcal{P}_0$ are non-conflicting. □

Lemma 5. *Let $\mathcal{P}'$ be a PPT protocol and $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ be a PPT adversary. Let $\mathcal{A}_{\mathcal{P}} = \text{ConstrAdv}(\mathcal{P}', \mathcal{A}_{\mathcal{P} \cup \mathcal{P}'})$ defined in Algorithm 4; then $\mathcal{A}_{\mathcal{P}}$ is a PPT adversary s.t. for any PPT protocol $\mathcal{P}$, the distribution of transcripts of executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ is identical to the distribution of transcripts of executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$.*

Proof. Let us first observe that $\mathcal{A}_{\mathcal{P}}$ is a PPT; this follows immediately from Algorithm 4, since $\mathcal{A}_{\mathcal{P}}$ either invokes an operation of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ or an operation of $\mathcal{P}'$. Since $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ and $\mathcal{P}'$ are both PPT, then $\mathcal{A}_{\mathcal{P}}$ is also PPT.

We will show that for all instances (which we categorize into one of three cases), the same algorithms are run to get outputs and updated states of instances, which are returned to the execution process, in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ as in executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$.

In the first case, the instance's subprotocol is included in $\mathcal{P}$, so it is also included in $(\mathcal{P} \cup \mathcal{P}')$. In this case, in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$, the execution process (Algorithm 1) runs $\mathcal{P}$, while in executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$, the execution process runs $(\mathcal{P} \cup \mathcal{P}')$. From Definition 6, for any subprotocol $sp \in \{0,1\}^*$ which is included in $\mathcal{P}$, for any operation name $op_name \in \{0,1\}^*$, $\mathcal{P}[sp, op_name] = (\mathcal{P} \cup \mathcal{P}')[sp, op_name]$. It follows that for these instances, the same algorithms are run in both types of executions.

In the second case, the instance's subprotocol is not included in $\mathcal{P}$ but is included in $\mathcal{P}'$, so it is also included in $(\mathcal{P} \cup \mathcal{P}')$. In this case, in executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$, the execution process runs $(\mathcal{P} \cup \mathcal{P}')$, while in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$, the execution process runs the RUN operation of $\mathcal{A}_{\mathcal{P}}$. In this case, in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$, from Algorithm 4, it follows that $\mathcal{A}_{\mathcal{P}}$ runs $\mathcal{P}'$. From Definition 6, for any subprotocol $sp \in \{0,1\}^*$

which is not included in $\mathcal{P}$ but is included in $\mathcal{P}'$, for any operation name $op_name \in \{0,1\}^*$, $\mathcal{P}'[sp, op_name] = (\mathcal{P} \cup \mathcal{P}')[sp, op_name]$. Thus, for these instances, the same algorithms are run in both types of executions.

In the third case, the instance's subprotocol is not included in $\mathcal{P}$ and is not included in $\mathcal{P}'$. In this case, in executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$, the execution process runs the RUN operation of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$, while in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$, the execution process runs the RUN operation of $\mathcal{A}_{\mathcal{P}}$. In this case, in executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$, from Algorithm 4, it follows that $\mathcal{A}_{\mathcal{P}}$ runs the RUN operation of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$. Thus, for these instances, the same algorithms are run in both types of executions.

It follows that in an execution of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ and in an execution of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$, the same algorithms are run in the same cases to get outputs and updated states of instances, which are returned to the execution process. Consequently, an execution of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ is equivalent (in terms of the execution transcript) to an execution of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$.

Note that since $\mathcal{P}$ and $\mathcal{P}'$ are probabilistic (coin-tossing), then any specific execution of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ may produce a different transcript than any specific execution of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$. However, the probability that an execution of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$ produces any specific transcript T is equal to the probability that an execution of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$ produces the same transcript T. This implies that transcripts from executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$ have the same distribution as transcripts from executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$. □

Theorem 1 (Composition preserves requirement satisfaction)**.** *Let $\mathcal{P}$ be a PPT protocol and $\mathcal{R}$ be a requirement. Then, for any model $\mathcal{M}$, any execution operations X, and any PPT protocol $\mathcal{P}'$, for* TYPE $\in \{\infty, \text{POLY}\}$*, holds:*

$$\mathcal{P} \models^{\mathcal{M}}_{X,\text{TYPE}} \mathcal{R} \implies (\mathcal{P} \cup \mathcal{P}') \models^{\mathcal{M}}_{X,\text{TYPE}} \mathcal{R} \tag{2}$$

Proof. We will prove Eq. (2) by showing that:

$$(\mathcal{P} \cup \mathcal{P}') \not\models^{\mathcal{M}}_{X} \mathcal{R} \implies \mathcal{P} \not\models^{\mathcal{M}}_{X} \mathcal{R} \tag{7}$$

By Definition 5, the LHS of Eq. (7) implies that there exists some adversary $\mathcal{A}_{(\mathcal{P} \cup \mathcal{P}')} \in PPT$ s.t. $\mathcal{A}_{(\mathcal{P} \cup \mathcal{P}')} \models^{X}_{\text{asym}} \mathcal{M}$ and for some $(\pi, \beta) \in \mathcal{R}$:

$$\mathcal{A}_{(\mathcal{P} \cup \mathcal{P}')}, (\mathcal{P} \cup \mathcal{P}') \not\models_{X} (\pi, \beta) \tag{8}$$

Since $\mathcal{P}' \in PPT$, then for any protocol $\mathcal{P}^* \in PPT$: $(\mathcal{P}^* \cup \mathcal{P}') \in PPT$. Since $\mathcal{A}_{(\mathcal{P} \cup \mathcal{P}')} \models_{X} \mathcal{M}$, by Definition 4, this implies that $\forall \mathcal{P}^* \in PPT$:

$$\forall (\pi, \beta') \in \mathcal{M} : \mathcal{A}_{(\mathcal{P} \cup \mathcal{P}')}, (\mathcal{P}^* \cup \mathcal{P}') \models_{X} (\pi, \beta') \tag{9}$$

Using Lemma 5, there exists an adversary $\mathcal{A}_{\mathcal{P}} \in PPT$ such that for any PPT protocol $\mathcal{P}^*$, transcripts from executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P}^* \cup \mathcal{P}')$ have the same distribution as transcripts from executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}^*$. This implies that:

$$\forall \mathcal{P}^* \in PPT : \forall (\pi, \beta') \in \mathcal{M} : \mathcal{A}_{\mathcal{P}}, \mathcal{P}^* \models_{X} (\pi, \beta') \tag{10}$$

Thus, by Definition 4, $\mathcal{A}_{\mathcal{P}} \models^{X}_{\text{asym}} \mathcal{M}$.

Using the same adversary $\mathcal{A}_{\mathcal{P}}$ and letting $\mathcal{P}^* = \mathcal{P}$, we get that transcripts from executions of $\mathcal{A}_{\mathcal{P} \cup \mathcal{P}'}$ against $(\mathcal{P} \cup \mathcal{P}')$ have the same distribution as transcripts from executions of $\mathcal{A}_{\mathcal{P}}$ against $\mathcal{P}$. From this and Eq. (8), it follows that for some $(\pi, \beta) \in \mathcal{R}$:

$$\mathcal{A}_{\mathcal{P}}, \mathcal{P} \not\models_{X} (\pi, \beta) \tag{11}$$

Since $(\pi, \beta) \in \mathcal{R}$, then Eq. (11) and Definition 5 imply the RHS of Eq. (7). □

E.2. Proofs of the Blackbox Composition Lemmas

We present proofs of the two blackbox composition lemmas from §4.2.

Lemma 6 (Monolithic black-box composition lemma)**.** *Let $\mathcal{P}$, $\mathcal{P}'$ be non-conflicting PPT protocols, X be execution operations, and $\mathcal{R}$, $\mathcal{M}$, $\mathcal{R}'$, $\mathcal{M}'$ be sets of specifications. Then, for* TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\left[\mathcal{P} \models^{\mathcal{M}, \mathcal{R}'}_{X,\text{TYPE}} \mathcal{R}\right] \wedge \left[\mathcal{P}' \models^{\mathcal{M}'}_{X,\text{TYPE}} \mathcal{R}'\right] \implies (\mathcal{P} \cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\text{TYPE}} \mathcal{R} \tag{5}$$

Proof. Starting with the 2nd term of the LHS of Eq. (5), i.e., $\mathcal{P}' \models^{\mathcal{M}'}_{X,\text{TYPE}} \mathcal{R}'$, it follows that:

$$\begin{aligned}
&(\mathcal{P}' \cup \mathcal{P}) \models^{\mathcal{M}'}_{X,\text{TYPE}} \mathcal{R}' && \text{[From Theorem 1]}\\
&\Rightarrow (\mathcal{P} \cup \mathcal{P}') \models^{\mathcal{M}'}_{X,\text{TYPE}} \mathcal{R}' && \text{[From Lemma 3]}\\
&\Rightarrow (\mathcal{P} \cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\text{TYPE}} \mathcal{R}' &&\\
&\Rightarrow (\mathcal{P} \cup \mathcal{P}') \models^{(\mathcal{M} \cup \mathcal{M}')}_{X,\text{TYPE}} \mathcal{R} && \text{[From 1st term of Eq. (5) LHS and Definition 8]}
\end{aligned}$$

□

Lemma 7 (Hybrid black-box composition lemma)**.** *Let $\mathcal{P}_1$, $\mathcal{P}_2$ be non-conflicting PPT protocols, X be execution operations, and $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2, \mathcal{M}_1, \mathcal{M}_2$ be sets of specifications. Then, for* TYPE $\in \{\infty, \text{POLY}\}$*:*

$$\left[\mathcal{P}_2 \models^{\mathcal{M}_2, \mathcal{R}_1}_{X,\text{TYPE}} \mathcal{R}_2\right] \wedge \left[\mathcal{P}_1 \models^{\mathcal{M}_1, \mathcal{R}_0}_{X,\text{TYPE}} \mathcal{R}_1\right] \implies (\mathcal{P}_2 \cup \mathcal{P}_1) \models^{(\mathcal{M}_2 \cup \mathcal{M}_1), \mathcal{R}_0}_{X,\text{TYPE}} \mathcal{R}_2 \tag{6}$$

Proof. Let $\mathcal{P}_0$ be any PPT protocol and $\mathcal{M}_0$ be any model such that:

$$((\mathcal{P}_2 \cup \mathcal{P}_1) \cup \mathcal{P}_0) \models^{((\mathcal{M}_2 \cup \mathcal{M}_1) \cup \mathcal{M}_0)}_{X,\text{TYPE}} \mathcal{R}_0$$

Then, it follows that:

$$(\mathcal{P}_1 \vec{\cup} (\mathcal{P}_2 \vec{\cup} \mathcal{P}_0)) \models^{(\mathcal{M}_1 \cup (\mathcal{M}_2 \cup \mathcal{M}_0))}_{X,\text{TYPE}} \mathcal{R}_0 \quad \text{[From Lemmas 1 and 3]}$$

$$\Rightarrow (\mathcal{P}_1 \vec{\cup} (\mathcal{P}_2 \vec{\cup} \mathcal{P}_0)) \models^{(\mathcal{M}_1 \cup (\mathcal{M}_2 \cup \mathcal{M}_0))}_{X,\text{TYPE}} \mathcal{R}_1 \quad \text{[From 2nd term of Eq. (6) LHS and Definition 8]}$$

$$\Rightarrow (\mathcal{P}_2 \vec{\cup} (\mathcal{P}_1 \vec{\cup} \mathcal{P}_0)) \models^{(\mathcal{M}_2 \cup (\mathcal{M}_1 \cup \mathcal{M}_0))}_{X,\text{TYPE}} \mathcal{R}_1 \quad \text{[From Lemmas 1 and 3]}$$

$$\Rightarrow (\mathcal{P}_2 \vec{\cup} (\mathcal{P}_1 \vec{\cup} \mathcal{P}_0)) \models^{(\mathcal{M}_2 \cup (\mathcal{M}_1 \cup \mathcal{M}_0))}_{X,\text{TYPE}} \mathcal{R}_2 \quad \text{[From 1st term of Eq. (6) LHS and Definition 8]}$$

$$\Rightarrow ((\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \vec{\cup} \mathcal{P}_0) \models^{((\mathcal{M}_2 \cup \mathcal{M}_1) \cup \mathcal{M}_0)}_{X,\text{TYPE}} \mathcal{R}_2 \quad \text{[From Lemmas 1 and 3]}$$

Since this holds for any PPT protocol $\mathcal{P}_0$ and any model $\mathcal{M}_0$ such that:

$$((\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \vec{\cup} \mathcal{P}_0) \models^{((\mathcal{M}_2 \cup \mathcal{M}_1) \cup \mathcal{M}_0)}_{X,\text{TYPE}} \mathcal{R}_0$$

Then it follows that:

$$(\mathcal{P}_2 \vec{\cup} \mathcal{P}_1) \models^{X, (\mathcal{M}_2 \cup \mathcal{M}_1), \mathcal{R}_0}_{\text{asym}} \mathcal{R}_2 \quad \text{[From Definition 8]}$$

□

Appendix F.
Interaction Between Instances: the Ask Convention and Model

The ask model specification $\mathcal{M}^{\Delta_{\text{ASK}}, Fields}_{\text{ASK}}$ is shown in Algorithm 13. The predicate $\mathcal{M}^{\Delta_{\text{ASK}}, Fields}_{\text{ASK}}.\pi$ (shown in Algorithm 13 in §4.3) uses the functions defined in Algorithms 14 to 16.

The ask model enforces some rules, precisely defined in Algorithm 13 and Algorithms 14 to 16, including rules about parent instances, child instances, and the values of the 'calling_sp' and 'calling_ins' input fields; we highlight some of these rules here. An output containing an ask request to invoke an instance which is neither a new instance, nor the existing parent, nor an existing child, is entirely ignored. Similarly an output containing an ask request to invoke an operation intended to be invoked by a different subprotocol, according to the operation naming convention described above, is entirely ignored. The values of the 'calling_sp' and 'calling_ins' input fields must always be legitimate, i.e., either the existing parent, an existing child, or the non-existing parent of the instance.[11] In the cases of the existing parent or an existing child, the event must be the correct invocation matching an ask request previously outputted by the parent or by the child, respectively. In the case of the non-existing parent, the restrictions are described in Algorithm 16. One of these restrictions is that the specified calling instance must not exist during the entire execution, which prevents a 'parent' from being created after 'its child'. Figure 6 in §F illustrates the ask convention and model.

11. This is a limitation, for simplicity. Supporting instances which can be invoked by multiple other non-child instances would be useful, but it would require care to prevent 'unauthorized' calling instances. Without proper restrictions, the adversary could, e.g., specify an adversarial instance as the calling instance of an unrelated benign instance, which may later invoke the adversarial instance and expose sensitive information to the adversary.

F.0.1. Preventing Eavesdropping. Secret values such as a private keys or secrets, may be passed between instances of subprotocols, e.g., when a key exchange protocol outputs a key to a protocol using the key, or when a secret value such as a deciphered message is outputted to another protocol.

Without any restrictions, the adversary could eavesdrop on any output field, by including the field name in the past outputs set *pastouts* in an event at an adversarial instance, as described in §2.2.

The ask model prevents eavesdropping by child instances with an existing parent, and by parent instances being invoked by child instances. This is because according to the $\mathcal{M}^{\Delta_{\text{ASK}}, Fields}_{\text{ASK}}.\pi$ model predicate, if the adversary identifies an 'Input' event as an ASK invocation, then the event must be a valid ASK invocation corresponding to an earlier valid ASK request, so only the fields relevant to the ASK request can be passed as part of the input, and no additional fields can be eavesdropped on.

In addition, the $\mathcal{M}^{\Delta_{\text{ASK}}, Fields}_{\text{ASK}}$ model prevents eavesdropping by instances with non-existing parents (i.e., invoked directly by the adversary) on output fields specified in the parameter $Fields$ if they are part of valid ASK requests. Consequently, if the ask model is assumed, then the adversary is allowed to eavesdrop only on output fields which are either not part of valid ASK requests, not included in the set $Fields$, or are part of valid ASK requests to invoke adversarial instances.

Appendix G.
Inter-Entity Communication: the COM Convention and Model

We expect many or most applications of CMoSS to involve distributed algorithms and protocols, involving multiple communicating entities. In practice, this will require support for some underlying communication mechanism. CMoSS can support different communication mechanisms, with different properties, enforced by appropriate models.

For example, the simple authenticated request response (ARR) protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ of Algorithm 22 assumes an underlying bounded-delay communication mechanism. We model the interaction between ARR and the communication mechanism using the *COM convention*, which is similar to the *ask convention* presented in §4.3 with different expected behavior, which is enforced by the $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ model. We define a protocol using the COM convention as if it will be composed with a COM subprotocol, even though we do not intend at first to compose it with a COM subprotocol, but instead assume the $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ model.

In the COM convention, the invoking instance outputs an ASK field which is in the same format as in the ask convention, with subprotocol identifier 'COM' (e.g., line 21 in Algorithm 22). So far, this is only a special case of the ask convention, which *allows* the subsequent implementation of

Algorithm 13 ASK model $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}$, used with the Ask convention for composition of protocols

1: **Allowed failure probability** $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.\beta \equiv 0$
2: **Subprotocols** $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.Sp \equiv \emptyset$
3: **Model predicate** $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.\pi(T)$
4: **return** $\top$ **iff**:
5: $\forall e \in T$ **s.t.** $\left[\begin{array}{l} e.\tau + \Delta_{\text{ASK}} \leq \max\{e'.\tau \mid e' \in T\} \\ \wedge\ \text{VALIDASKREQUEST}(T, e) \end{array} \right]$: // *For every event which is at least* Δ_{ASK} *time before the end of the execution and outputs a valid* ASK *request (see Alg. 14)*
6: $\exists e' \in T$ **s.t.** $\text{VALIDASKINVOCATION}(T, e, e', \Delta_{\text{ASK}})$ // *There is an event within* Δ_{ASK} *time in which the specified instance is invoked according to the* ASK *request (see Alg. 15)*
7: $\wedge\ \forall e \in T$ **s.t.** $\left[\begin{array}{l} e.EventType = \text{'Input'} \\ \wedge\ e.vals_{\mathcal{A}}.\text{ASK}.calling_\tau \neq \perp \end{array} \right]$: // *And for every 'Input' event which the adversary identified as an* ASK *invocation*
8: $\exists e' \in T$ **s.t.** $\left[\begin{array}{l} \text{VALIDASKREQUEST}(T, e') \\ \wedge\ \text{VALIDASKINVOCATION}(T, e', e, \Delta_{\text{ASK}}) \end{array} \right]$ // *The event is a valid* ASK *invocation (see Alg. 15) for an earlier valid* ASK *request (see Alg. 14)*
9: $\wedge\ \nexists e' \in T$ **s.t.** $\left[\begin{array}{l} e' \neq e\ \wedge\ e'.EventType = \text{'Input'} \\ \wedge\ e'.vals_{\mathcal{A}}.\text{ASK}.calling_\tau \\ \quad = e.vals_{\mathcal{A}}.\text{ASK}.calling_\tau \end{array} \right]$ // *And there is no other 'Input' event which is an invocation for the same* ASK *request*
10: $\wedge\ \forall e \in T$ **s.t.** $\left[\begin{array}{l} e.EventType = \text{'Input'} \\ \wedge\ e.vals_{\mathcal{A}}.\text{ASK}.calling_\tau = \perp \end{array} \right]$: // *And for every 'Input' event which adversary did not identify as an* ASK *invocation*
11: $\text{VALIDINVOCATIONBYADVERSARY}(T, e, Fields)$ // *The event is a valid invocation by the adversary (see Alg. 16), where the calling instance is the 'non-existing parent'*
12: **end model predicate**

the communication mechanism as a subprotocol; however, we do *not* present a corresponding COM subprotocol, and therefore do not analyze $\mathcal{P}_{\text{ARR}}^{\mathcal{S}}$ assuming blackbox requirements for COM.[12]

Instead, we prove the security of $\mathcal{P}_{\text{ARR}}^{\mathcal{S}}$ under a *model* we denote as $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$.[13] Intuitively, the $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$ model ensures timely delivery of sent payloads with correct sender network addresses. The model provides the adversary with both eavesdropping and off-path (spoofing) capabilities[14]. In a bit more detail, the model ensures:

- **Bounded delay initialization.** If instance (ent, sp, i), asks to invoke operation INIT at any COM instance, then, within at most Δ_{com} time, the adversary invokes operation COM-INITRESP at (ent, sp, i), inputting a value $addr_i$, which we call the *address* assigned to (ent, sp, i). An instance is assigned only one address (if any).[15] Not all instances have an address; for example, COM instances do not have an address. In addition, the model ensures that there is only one instance per every assigned address, i.e., assigned addresses are unique.
- **Bounded delay communication.** Suppose that at time t, instance (ent', sp', j) asks to invoke operation SENDPKT at any COM instance, with parameters PAYLOAD:p and DSTADDR:$addr_{DST}$, and that $addr_{DST}$ was assigned to instance (ent, sp, i). Then, within at most Δ_{com} time, the adversary invokes operation COM-RCVPKT at instance (ent, sp, i), with parameters PAYLOAD:p and SRCADDR:$addr_{SRC}$, where $addr_{SRC}$ is the address assigned to (ent', sp', j)[16].

Other network-adversary models can be similarly defined, e.g., an eavesdropper-only model, or a model allowing

12. Alternatively, or in a later iteration of designing the protocol, a bounded-delay communication black-box requirement could be used, instead of the $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$ model. This would make it convenient to compose the protocol with a COM subprotocol.

13. Note that if both the $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}$ model and $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$ model are assumed, as in Section 5, then the adversary still invokes COM instances when asked to do so, due to the $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}$ model. In addition, in order to ensure communication as required by the $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$ model, the adversary may have to output appropriate ASK requests by (adversarial) COM instances, in order to be able to invoke operations starting with 'COM-' while still satisfying the $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}$ model.

14. More precisely, the model $\mathcal{M}_{\text{COM}}^{\Delta_{com}}$ does not prevent these capabilities, which are allowed by default in CMoSS.

15. We limit to one address per instance for simplicity. In particular, with this limitation, when an instance asks to send a packet (Line 21 in Algorithm 22), it does not need to specify which one of its addresses it wants to use as the source address, and when an instance receives a packet (Line 23 in Algorithm 22), it does not need to receive a destination address to determine for which one of its addresses the packet is intended. Another communication model could be defined to support multiple addresses per instance.

16. If no address was assigned to (ent', sp', j) so far, then it would be assigned now.

Algorithm 14 The VALIDASKREQUEST predicate used by the ASK model predicate $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.\pi$ (Algorithm 13).

procedure VALIDASKREQUEST(T, e)
  return $\top$ **iff**:
    $e.EventType =$ 'Run' // *e is a 'Run' event*
    $\wedge\ e.out.\text{ASK} \neq \perp$ // *e outputted an* ASK *request*
    // *The operation name in the* ASK *request is meant to be invoked by parent instances (i.e., does not include a hyphen) or by instances of subprotocol $e.sp$ (i.e., starts with $e.sp$-)*
    $\wedge \left[\begin{array}{l} \text{'-' is not in } e.out.\text{ASK}.op_name \\ \vee\ \exists str \in \{0,1\}^* \textbf{ s.t. } e.out.\text{ASK}.op_name = e.sp\text{-}str \end{array} \right]$
    // *The* ASK *request is to invoke a new instance, a valid child instance, or the valid parent instance*
    $\wedge \left[\begin{array}{l} e.out.\text{ASK}.ins = \perp \\ \vee\ \text{VALIDPARENTCHILD}\left(\begin{array}{c} T, e.ent, \\ e.sp, e.ins, \\ e.out.\text{ASK}.sp, e.out.\text{ASK}.ins \end{array} \right) \\ \vee\ \text{VALIDPARENTCHILD}\left(\begin{array}{c} T, e.ent, \\ e.out.\text{ASK}.sp, e.out.\text{ASK}.ins, \\ e.sp, e.ins \end{array} \right) \end{array} \right]$
end procedure

procedure VALIDPARENTCHILD($T, ent, sp_{\text{p}}, ins_{\text{p}}, sp_{\text{c}}, ins_{\text{c}}$)
  return $\top$ **iff**:
    // *The parent instance asked to invoke a new instance of the child subprotocol, and there was a corresponding* ASK *invocation at the child instance.*
    $\exists e_{\text{p}}, e_{\text{c}} \in T$ **s.t.** $\left[\begin{array}{l} e_{\text{p}}.ent = ent\ \wedge\ e_{\text{p}}.sp = sp_{\text{p}}\ \wedge\ e_{\text{p}}.ins = ins_{\text{p}} \\ \wedge\ e_{\text{p}}.EventType = \text{'Run'} \\ \wedge\ e_{\text{p}}.out.\text{ASK}.sp = sp_{\text{c}}\ \wedge\ e_{\text{p}}.out.\text{ASK}.ins = \perp \\ \wedge\ e_{\text{c}}.ent = ent\ \wedge\ e_{\text{c}}.sp = sp_{\text{c}}\ \wedge\ e_{\text{c}}.ins = ins_{\text{c}} \\ \wedge\ e_{\text{c}}.EventType = \text{'Input'} \\ \wedge\ e_{\text{c}}.vals_{\mathcal{A}}.calling_\tau = e_{\text{p}}.\tau \end{array} \right]$
end procedure

MitM capabilities at specific times or entities.

Appendix H. Composite Clock-Synchronization Protocol

Shorthand: all operations receive as input the instance's state s, the local clock value clk, and the input fields $inp.calling_sp$ and $inp.calling_ins$, and return an updated state. To reduce clutter, we do not write these values explicitly in the text below.

H.1. Clock Synchronization

H.1.1. Operations of a Clock Synchronization Protocol (CS Scheme). There are only two operations that every clock synchronization (CS) protocol must support, although, of course, protocols would usually use additional operations to fulfill the requirements:

- INIT. Inputs: role $role$, server's address $SrvAddr$ and server's public key $SrvPK$; the server's address and public key are used only when initializing clients (and ignored when initializing a server). Outputs to parent: none ($\perp$) if $role =$ 'client'; server's address $addr$ and public key PK if $role =$ 'server'.
- GETTIME. No inputs. Outputs to parent: time value.

These operations, as all operations in the CMoSS execution, are invoked due to instructions from the adversary to the execution process; normally, the adversary instructs to invoke operations, after being asked to do so by some *parent instance* which has a clock synchronization *child instance*, typically, an application using the clock synchronization protocol.

Similarly, the clock synchronization protocol has to ask the adversary to invoke the parent instance in order to provide the output of a completed operation. We next define operations that the parent instance should implement, to receive these outputs (in addition to possibly implementing

Algorithm 15 The VALIDASKINVOCATION predicate used by the ASK model predicate $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.\pi$ (Algorithm 13).

procedure VALIDASKINVOCATION(T, e, e', Δ)
  return $\top$ **iff**:
    $e.\tau < e'.\tau \leq e.\tau + \Delta$ // *Event e' is after e, but not more than Δ time after*
    $\wedge\ e'.EventType =$ 'Input' // *e' is a 'Input' event*
    $\wedge\ e'.vals_{\mathcal{A}}.calling_\tau = e.\tau$ // *e' is an* ASK *invocation for the* ASK *request outputted in e*
    // *Events e' and e are at the same entity, and the subprotocol in e' is the subprotocol specified in the* ASK *request in e*
    $\wedge\ e'.ent = e.ent\ \wedge\ e'.sp = e.out.\text{ASK}.sp$
    // *e' is not an X-operation event, and the operation name is the one specified in the* ASK *request in e*
    $\wedge\ e'.Xopr \neq \top\ \wedge\ e'.op_name = e.out.\text{ASK}.op_name$
    // *The direct input fields in e' include only the 'calling_sp' field, with the subprotocol of e as the value, and the 'calling_ins' field, with the instance identifier of e as the value*
    $\wedge\ e'.inp = \{(\text{'calling_sp'}, e.sp), (\text{'calling_ins'}, e.ins)\}$
    // *The past output fields added to the input in e' include only the output fields from e specified in the $Params$ field of the* ASK *request in e*
    $\wedge\ e'.pastouts = \{(field, e.\tau) \mid field \in e.out.\text{ASK}.Params\}$
    // *If the* ASK *request is to invoke an existing instance, then the instance identifier in e' is the one specified in the* ASK *request in e. If the* ASK *request is to invoke a new instance, then the instance identifier in e' is not $\perp$ and was not used for any instance of the specified subprotocol at the same entity in any earlier event.*
    $\wedge \left[\begin{array}{l} [e.out.\text{ASK}.ins \neq \perp\ \wedge\ e'.ins = e.out.\text{ASK}.ins] \\ \vee \left[\begin{array}{l} e.out.\text{ASK}.ins = \perp\ \wedge\ e'.ins \neq \perp \\ \wedge\ \nexists e'' \in T \textbf{ s.t. } \left[\begin{array}{l} e''.ins = e'.ins\ \wedge\ e''.ent = e'.ent \\ \wedge\ e''.sp = e'.sp\ \wedge\ e''.\tau < e'.\tau \end{array} \right] \end{array} \right] \end{array} \right]$
end procedure

other operations):

- CS-INITRESPSERVER. Inputs: server's address $addr$ and public key PK. Should be implemented by a parent instance creating a child clock synchronization *server* instance.
- CS-INITRESPCLIENT. No inputs. Should be implemented by a parent instance creating a child clock synchronization *client* instance.
- CS-GETTIMERESP. Inputs: a time value $time$. Should be implemented by a parent instance with any child clock synchronization instance.

H.1.2. The Clock Synchronization Requirement $\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}}$. A clock synchronization protocol is required to provide time values with bounded offset from real time, and the service should be provided within bounded time. In addition, initialization of a clock synchronization server or client should take bounded time. For convenience, we use the same bound of Δ_{CS} for all three bounds (on the offset of returned time values, on the delay until the time values are returned, and on the initialization time).[17] The clock synchronization requirement $\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}}$ includes the three requirements described below.

- **Bounded-offset time responses ($\mathcal{R}_{\text{CS-OFFSET}}^{\Delta_{\text{CS}}}$, Algorithm 17).** The maximal offset between the time response outputted by a benign, initialized CS instance (to operation 'CS-GetTimeResp') and the time of the outputting event is at most Δ_{CS}.
- **Timely time responses ($\mathcal{R}_{\text{CS-DELAY}}^{\Delta_{\text{CS}}}$, Algorithm 18).** The delay between giving input to the 'GetTime' operation at a benign, initialized CS instance and the instance outputting the time response is at most Δ_{CS}. (No response is required if the execution ends within less than Δ_{CS} time.)
- **Timely initialization ($\mathcal{R}_{\text{CS-INIT}}^{\Delta_{\text{CS}}}$).** The delay between giving input to the 'Init' operation at a benign, previously uninitialized CS server or client instance and the instance completing initialization is Δ_{CS}. (No response

17. The three bounds could be separated. They may have much different values. For example, in the protocol $\mathcal{P}_{\text{UnauthCS}}$, the initialization delay would likely be much greater than the delay to get time values. This is because, in $\mathcal{P}_{\text{UnauthCS}}$, initialization of a CS instance involves initializing an ARR instance, while getting a time only requires adding two values and returning their sum.

Algorithm 16 The VALIDINVOCATIONBYADVERSARY predicate used by the ASK model predicate $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}.\pi$ (Algorithm 13).

procedure VALIDINVOCATIONBYADVERSARY(T, e, F)
  $E \leftarrow \{e' \mid e'.ent = e.ent \ \wedge \ e'.sp = e.sp \ \wedge \ e'.ins = e.ins\}$
  $e'' \leftarrow$ event in E **s.t.** $e''.\tau = \min\{e'.\tau \mid e' \in E\}$
  return $\top$ **iff**:
    // The 'calling_sp' and 'calling_ins' input fields in e are the same as in the first event at the same entity
    $e.inp.calling_sp = e''.inp.calling_sp$
    $\wedge \ e.inp.calling_ins = e''.inp.calling_ins$
    // There is no event in T at the same entity as e with subprotocol and instance identifiers equal to those in the 'calling_sp' and 'calling_ins' input fields in e, respectively
    $\wedge \nexists e' \in T$ **s.t.** $\begin{bmatrix} e'.ent = e.ent \\ \wedge \ e'.sp = e.inp.calling_sp \\ \wedge \ e'.ins = e.inp.calling_ins \end{bmatrix}$
    // If e is not an X-operation event, then the operation name is meant to be invoked by parent instances (i.e., does not include a hyphen)
    $\wedge \ [e.Xopr = \top \ \vee$ '-' is not in $e.op_name]$
    // Any past output field added to the input in e must either be a field from an event without a valid ASK *request (see Alg. 14), be a field from an event in which the field is not specified in the $Params$ field of the* ASK *request, or be a field which is not in the set F*
    $\wedge \ \forall (\textit{field}, \tau') \in e.\textit{pastouts}, e' \in T$ **s.t.** $e'.\tau = \tau' :$
      $\begin{bmatrix} \vee \ \neg\text{VALIDASKREQUEST}(T, e') \\ \vee \ \textit{field} \notin e'.out.\text{ASK}.Params \\ \textit{field} \notin F \end{bmatrix}$
end procedure

is required if the execution ends within less than Δ_{CS} time.) The requirement $\mathcal{R}_{sp\text{-INIT}}^{\Delta}$ (Algorithm 19) is used, with parameter values $sp = \text{CS}$ and $\Delta = \Delta_{\text{CS}}$.

Using the above requirements, the clock synchronization requirement $\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}}$ is defined as:

$$\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}} \equiv \mathcal{R}_{\text{CS-OFFSET}}^{\Delta_{\text{CS}}} \cup \mathcal{R}_{\text{CS-DELAY}}^{\Delta_{\text{CS}}} \cup \mathcal{R}_{\text{CS-INIT}}^{\Delta_{\text{CS}}}$$

The $\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}}$ requirement does not require GETTIME responses to identify the corresponding requests, so it does not support concurrent GETTIME requests. If some application wants to invoke GETTIME at the same CS instance multiple times concurrently, then it may require a different requirement for the CS subprotocol.

H.1.3. The $\mathcal{P}_{\text{UnauthCS}}$ Clock Synchronization Protocol. We describe how the protocol $\mathcal{P}_{\text{UnauthCS}}$ (Algorithm 20) works, and then we prove that $\mathcal{P}_{\text{UnauthCS}}$ satisfies $\mathcal{R}_{\text{CS}}^{\Delta_{\text{CS}}}$, assuming $\mathcal{M}_{\text{CS}}^{\Delta_{\text{CS}}}$ and a blackbox satisfying $\mathcal{R}_{\text{ARR}}^{\Delta_{\text{ARR}}}$; the analysis bounds Δ_{CS} using the time-bounds of $\mathcal{M}_{\text{CS}}^{\Delta_{\text{CS}}}$ and $\mathcal{R}_{\text{ARR}}^{\Delta_{\text{ARR}}}$. We use the ask convention (§4.3) when describing $\mathcal{P}_{\text{UnauthCS}}$.

The CS protocol first needs to initialize an ARR instance. In the simplified pseudocode, we wrote that the CS protocol 'asks' to initialize a new instance of subprotocol ARR (Line 4). More precisely, the output of the requesting operation (CS.INIT) includes a 'key-value pair' of the form ('ASK', ('ARR', 'Init', $\perp$, $Params$)), where $Params$ = ('role', 'SrvAddr', 'SrvPK'). The 'ASK' keyword identifies this output as a request asking the adversary to invoke a subprotocol, in this case, 'ARR'; the 'INIT' keyword identifies the operation to be invoked[18]. The third element in the ask value, which is here $\perp$, is the instance identifier of the instance to be invoked. A value of $\perp$ for the instance identifier indicates that this should be a new instance, which does not yet have an instance identifier. The set $Params$ contains the names of the input parameters to the ARR.INIT operation. The output of CS.INIT separately includes key-value pairs for each of the fields named in $Params$, e.g., ('role', $role$). In the simplified pseudocode, we omit the 'calling_sp' and 'calling_ins' input fields and the saving of their values to $s.parent_sp$ and $s.parent_ins$, respectively.

The pseudocode of $\mathcal{P}_{\text{UnauthCS}}$ states that the new ARR instance is 'identified as $s.ARRinstance$'. This is a simplification, designed to ease writing and improve readability of the pseudocode. What this simplified pseudocode means is that when the identifier is provided as input, according to $\mathcal{M}_{\text{ASK}}^{\Delta_{\text{ASK}},Fields}$, to the response operation to ARR.INIT, then the response operation saves the identifier in vari-

18. In our pseudocode, when we write 'initialize a (new) instance' (e.g., in Line 4 in Algorithm 20), we mean 'invoke operation INIT'.

Algorithm 17 The bounded-offset time responses requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-OFFSET}}$ for CS protocols

// If a benign, initialized CS instance outputs a time response, then the time value in the response is within Δ_{CS} of the current time

1: **Allowed failure probability** $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-OFFSET}}.\beta \equiv 0$
2: **Requirement predicate** $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-OFFSET}}.\pi(T)$
3: $e_{\text{Last}} \leftarrow$ event in T **s.t.** $e_{\text{Last}}.\tau = \max\{e'.\tau \mid e' \in T\}$ *// Get the last event e_{Last} of the execution*

// If adversary did not output the time of an event in $e_{Last}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$, then it does not win

4: **if** $\nexists e \in T$ **s.t.** $e.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ **then return** $\top$
5: $e_{\text{TimeResp}} \leftarrow$ event in T **s.t.** $e_{\text{TimeResp}}.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ *// Get the event specified by the adversary*

// The adversary does not win if the event is at a corrupt entity, is not at an instance of subprotocol CS, is at an uninitialized instance, is an X-operation event, or does not output a time response in the ASK *output field*

6: **if** $\left[\begin{array}{l} e_{\text{TimeResp}}.ent \in \mathsf{Corrupt}(T) \\ \vee\ e_{\text{TimeResp}}.sp \neq \text{`CS'} \\ \vee\ \neg\text{INITIALIZEDCSINSTANCE}(T, e_{\text{TimeResp}}) \\ \vee\ e_{\text{TimeResp}}.Xopr = \top \\ \vee\ e_{\text{TimeResp}}.out.\text{ASK}.op_name \neq \text{`CS-GetTimeResp'} \end{array}\right]$ **then return** $\top$

// The adversary wins if the time value in the ASK *output field is not within Δ_{CS} of the time of the event*

7: **if** $|e_{\text{TimeResp}}.out.\text{ASK}.Params.time - e_{\text{TimeResp}}.\tau| > \Delta_{\text{CS}}$ **then return** $\bot$
8: **return** $\top$ *// The adversary does not win*
9: **end requirement predicate**

// Return $\top$ if the instance in event e is an initialized CS instance

10: **procedure** INITIALIZEDCSINSTANCE(T, e)

// The instance in event e is an initialized CS instance if the subprotocol is 'CS' and the instance previously outputted an ASK *request to invoke operation 'CS-InitRespServer' or operation 'CS-InitRespClient'*

11: **if** $\exists e' \in T$ **s.t.** $\left[\begin{array}{l} e'.\tau < e.\tau \\ \wedge\ e'.ent = e.ent \\ \wedge\ e'.sp = e.sp = \text{`CS'} \\ \wedge\ e'.ins = e.ins \\ \wedge\ e'.out.\text{ASK}.op_name \in \{\text{`CS-InitRespServer', `CS-InitRespClient'}\} \end{array}\right]$ **then return** $\top$

12: **return** $\bot$ *// The instance is not an initialized CS instance*
13: **end procedure**

able $s.ARRinstance$. The protocol $\mathcal{P}_{\text{UnauthCS}}$ expects the response operation to be either CS.ARR-INITRESPSERVER or CS.ARR-INITRESPCLIENT.

For CS servers, initialization is completed once the ARR.INIT operation invokes the CS.ARR-INITRESPSERVER operation.

Notably, the CS.ARR-INITRESPSERVER operation asks to invoke operation CS-INITRESPSERVER of the *parent* instance. In this work, such a parent instance does not exist; however, $\mathcal{P}_{\text{UnauthCS}}$ can be composed in the future with other protocols, which can create instances of $\mathcal{P}_{\text{UnauthCS}}$, and which can include operation CS-INITRESPSERVER.

For CS clients, initialization is not yet complete when ARR.INIT invokes the CS.ARR-INITRESPCLIENT operation. After initializing an ARR instance, a client CS instance uses the SENDREQ operation of the ARR subprotocol to send a request to the CS server instance (Line 12). The ARR instance is identified by subprotocol 'ARR' (omitted in the pseudocode for simplicity) and instance identifier $s.ARRinstance$, which is kept by the CS instance in its state. A CS client completes its initialization only after receiving a response from the CS server and updating the client's offset value.

H.1.4. The $\mathcal{M}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}_{\textbf{CS}}$ Model and the $\mathcal{R}^{\Delta_{\text{ARR}}}_{\textbf{ARR}}$ Requirement. After the requirements of a protocol are defined, the next step in the design of a protocol is to identify design assumptions, i.e., the models and blackbox requirements. Some models and blackbox requirements may be initially 'missed' and only identified during protocol design and analysis, requiring iterations. Iterations are also used to improve the protocols, designing first with multiple/strong models, then improving by removing models, or weakening models, or replacing models by corresponding blackbox requirements, which we then implement by a subprotocol.

Clock synchronization can be quite challenging. To sim-

Algorithm 18 The timely time responses requirement $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-DELAY}}$ for CS protocols

// Outputting the time by a benign, initialized CS instance takes at most Δ_{CS} time; lack of response is allowed if less than Δ_{CS} time remains in the execution

Allowed failure probability $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-DELAY}}.\beta \equiv 0$

Requirement predicate $\mathcal{R}^{\Delta_{\text{CS}}}_{\text{CS-DELAY}}.\pi(T)$

$e_{\text{Last}} \leftarrow$ event in T **s.t.** $e_{\text{Last}}.\tau = \max\{e'.\tau \mid e' \in T\}$ *// Get the last event e_{Last} of the execution*

// If adversary did not output the time of an event in $e_{Last}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$, then it does not win

if $\nexists e \in T$ **s.t.** $e.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ **then return** $\top$

$e_{\text{GetTimeInput}} \leftarrow$ event in T **s.t.** $e_{\text{GetTimeInput}}.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ *// Get the event specified by the adversary*

// The adversary does not win if the event is less than Δ_{CS} time before the end of the execution

if $e_{\text{Last}}.\tau - e_{\text{GetTimeInput}}.\tau < \Delta_{\text{CS}}$ **then return** $\top$

// The adversary does not win if the event is at a corrupt entity, is not at an instance of subprotocol CS, is at an uninitialized instance, is an X-operation event, or is not an 'Input' event of operation 'GetTime'

if $\left[\begin{array}{l} e_{\text{GetTimeInput}}.ent \in \mathsf{Corrupt}(T) \\ \vee\ e_{\text{GetTimeInput}}.sp \neq \text{'CS'} \\ \vee\ \neg\textsc{InitializedCSInstance}(T, e_{\text{GetTimeInput}}) \\ \vee\ e_{\text{GetTimeInput}}.Xopr = \top \\ \vee\ e_{\text{GetTimeInput}}.EventType \neq \text{'Input'} \\ \vee\ e_{\text{GetTimeInput}}.op_name \neq \text{'GetTime'} \end{array}\right]$ **then return** $\top$

// The adversary wins if the instance did not output a time response in the ASK *output field within Δ_{CS} time after the 'Input' 'GetTime' event*

if $\nexists e_{\text{TimeResp}} \in T$ **s.t.** $\left[\begin{array}{l} e_{\text{TimeResp}}.ent = e_{\text{GetTimeInput}}.ent \\ \wedge\ e_{\text{TimeResp}}.sp = e_{\text{GetTimeInput}}.sp = \text{'CS'} \\ \wedge\ e_{\text{TimeResp}}.ins = e_{\text{GetTimeInput}}.ins \\ \wedge\ e_{\text{TimeResp}}.out.\text{ASK}.op_name = \text{'CS-GetTimeResp'} \\ \wedge\ e_{\text{GetTimeInput}}.\tau < e_{\text{TimeResp}}.\tau \leq e_{\text{GetTimeInput}}.\tau + \Delta_{\text{CS}} \end{array}\right]$ **then return** $\bot$

return $\top$ *// The adversary does not win*

end requirement predicate

plify, we assume a strong model, $\mathcal{M}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}_{\mathsf{CS}}$, and a strong blackbox requirement, $\mathcal{R}^{\Delta_{\text{ARR}}}_{\mathsf{ARR}}$.

The $\mathcal{M}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}_{\mathsf{CS}}$ model includes the $\mathcal{M}^{\text{IO}}_{\Delta_{io}}$ and $\mathcal{M}^{\Delta_{\text{ASK}},Fields}_{\text{ASK}}$ models, to ensure timely 'Run' events and timely interaction between $\mathcal{P}_{\text{UnauthCS}}$ and the blackbox ARR protocol (using the ASK convention), and three other models, described below:

- $\mathcal{M}_{\text{FIXEDOFF}}$, ensuring that the local clock of each entity has a fixed offset from the real time.
- $\mathcal{M}^{\Delta_{\text{SRVOFF}}}_{\text{SRVOFF}}$, ensuring that the offset of the local clocks of all CS servers from the real time is at most Δ_{SRVOFF}.
- $\mathcal{M}^{\text{CS}}_{\text{INIT}}$, ensuring that clients are initialized with client-initialization-parameters outputted by a CS server. By viewing the subprotocol (CS) as a parameter, this model can be reused by other client-server protocols.

Hence:

$$\mathcal{M}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}_{\mathsf{CS}} \equiv \mathcal{M}^{\text{IO}}_{\Delta_{io}} \cup \mathcal{M}^{\Delta_{\text{ASK}},Fields}_{\text{ASK}} \cup \mathcal{M}_{\text{FIXEDOFF}} \cup \mathcal{M}_{\text{SRVOFF}^{\Delta_{\text{SRVOFF}}}} \cup \mathcal{M}^{\text{CS}}_{\text{INIT}}$$

H.1.5. Operations of an Authenticated Request-Response Protocol (ARR Scheme). Every authenticated request-response (ARR) protocol must support three operations, defined as follows:

- INIT. Inputs: role *role*, server's address *SrvAddr*, and server's public key *SrvPK*. Outputs to parent: none ($\bot$) if $role =$ 'client'; server's address $addr$ and public key PK if $role =$ 'server'.
- SENDREQ. Inputs: request req and request identifier $reqID$. Does not invoke parent.
- SENDRESP. Inputs: response $resp$, request identifier $reqID$, and client address $ClAddr$. Does not invoke parent.

We define operations that a parent instance which uses an ARR child instance should implement, to receive outputs of the ARR instance to its parent, or to be notified when the ARR instance is done with an operation.

- ARR-INITRESPCLIENT. Inputs: none.
- ARR-INITRESPSERVER. Inputs: server's address $addr$ and public key PK.
- ARR-RCVRESP. Inputs: response $resp$ and request identifier $reqID$.
- ARR-RCVREQ. Inputs: request req, request identifier $reqID$, and client address $ClAddr$.

Algorithm 19 The timely initialization requirement $\mathcal{R}^{\Delta}_{sp\text{-INIT}}$

// Initializing a benign, previously uninitialized server or client instance of subprotocol sp takes at most Δ time; lack of complete initialization is allowed if less than Δ time remains in the execution

Allowed failure probability $\mathcal{R}^{\Delta}_{sp\text{-INIT}}.\beta \equiv 0$

Requirement predicate $\mathcal{R}^{\Delta}_{sp\text{-INIT}}.\pi(T)$

$e_{\text{Last}} \leftarrow$ event in T **s.t.** $e_{\text{Last}}.\tau = \max\{e'.\tau \mid e' \in T\}$ *// Get the last event e_{Last} of the execution*

// If adversary did not output the time of an event in $e_{Last}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$, then it does not win

if $\nexists e \in T$ **s.t.** $e.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ **then return** $\top$

$e_{\text{InitInput}} \leftarrow$ event in T **s.t.** $e_{\text{InitInput}}.\tau = e_{\text{Last}}.vals_{\mathcal{A}}.\tau_{\mathcal{A}}$ *// Get the event specified by the adversary*

// The adversary does not win if the event is less than Δ time before the end of the execution

if $e_{\text{Last}}.\tau - e_{\text{InitInput}}.\tau < \Delta$ **then return** $\top$

// The adversary does not win if the event is at a corrupt entity, is not at an instance of subprotocol sp, is at an instance which previously began initialization, is an X-operation event, is not an 'Input' event of operation 'Init', or does not have 'server' or 'client' in the 'role' input field

if $\left[\begin{array}{l} e_{\text{InitInput}}.ent \in \mathsf{Corrupt}(T) \\ \vee\ e_{\text{InitInput}}.sp \neq sp \\ \vee\ \textsc{PreviouslyBeganInitialization}(T, e_{\text{InitInput}}) \\ \vee\ e_{\text{InitInput}}.Xopr = \top \\ \vee\ e_{\text{InitInput}}.EventType \neq \text{'Input'} \\ \vee\ e_{\text{InitInput}}.op_name \neq \text{'Init'} \\ \vee\ e_{\text{InitInput}}.fullinp.role \notin \{\text{'server'}, \text{'client'}\} \end{array}\right]$ **then return** $\top$

// The adversary wins if the instance did not output an initialization response matching the role, in the ASK *output field, within Δ time after the 'Input' 'Init' event*

if $\nexists e_{\text{InitResp}} \in T$ **s.t.** $\left[\begin{array}{l} e_{\text{InitResp}}.ent = e_{\text{InitInput}}.ent \\ \wedge\ e_{\text{InitResp}}.sp = e_{\text{InitInput}}.sp = sp \\ \wedge\ e_{\text{InitResp}}.ins = e_{\text{InitInput}}.ins \\ \wedge \left[\begin{array}{l} \left[\begin{array}{l} e_{\text{InitInput}}.fullinp.role = \text{'server'} \\ \wedge\ e_{\text{InitResp}}.out.\text{ASK}.op_name = sp\text{'-InitRespServer'} \end{array}\right] \\ \vee \left[\begin{array}{l} e_{\text{InitInput}}.fullinp.role = \text{'client'} \\ \wedge\ e_{\text{InitResp}}.out.\text{ASK}.op_name = sp\text{'-InitRespClient'} \end{array}\right] \end{array}\right] \\ \wedge\ e_{\text{InitInput}}.\tau < e_{\text{InitResp}}.\tau \leq e_{\text{InitInput}}.\tau + \Delta \end{array}\right]$ **then return** $\bot$

return $\top$ *// The adversary does not win*

end requirement predicate

// Return $\top$ if the instance in event e had an 'Input' 'Init' event before event e

procedure $\textsc{PreviouslyBeganInitialization}(T, e)$

// The instance in event e had an 'Input' 'Init' event before event e

if $\exists e' \in T$ **s.t.** $\left[\begin{array}{l} e'.\tau < e.\tau \\ \wedge\ e'.ent = e.ent \\ \wedge\ e'.sp = e.sp \\ \wedge\ e'.ins = e.ins \\ \wedge\ e'.EventType = \text{'Input'} \\ \wedge\ e'.op_name = \text{'Init'} \end{array}\right]$ **then return** $\top$

return $\bot$ *// The instance did not previously have an 'Input' 'Init' event*

end procedure

H.1.6. The $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR}}$ Black-box Requirement. An authenticated request-response protocol is required to provide authenticated responses (although requests do not need to be authenticated), a bound on how long it takes to receive

Algorithm 20 The $\mathcal{P}_{\text{UnauthCS}}$ clock synchronization protocol

operation CS.INIT($s, clk, role$, *SrvAddr*, *SrvPK*)
  // Verify role not $\perp$ and instance not already initialized
  if $role = \perp$ **or** $s.role \neq \perp$ **then return** s
  $s.role \leftarrow role$ *// Remember role*
  return s and **ask to initialize** a (new) instance of subprotocol ARR, identified as *s.ARRinstance*, with (ROLE:$s.role$, SRVADDR:*SrvAddr*, SRVPK:*SrvPK*)
    // Servers ignore SrvAddr, SrvPK parameters
end operation

operation CS.ARR-INITRESPSERVER($s, clk, addr, PK$)
  return s and **ask to invoke** operation CS-INITRESPSERVER of the **parent** instance, with parameters (ADDR:$addr$, PK:PK)
end operation

operation CS.ARR-INITRESPCLIENT(s, clk)
  $s.ReqTime \leftarrow clk$ *// Store current local clock*
  $s.reqID \xleftarrow{\$} \{0,1\}^{|s.1^\kappa|}$ *// Randomly choose a request ID*
  return s and **ask to invoke** operation SENDREQ of instance $s.ARRinstance$ with parameters (REQ:'SENDCLK', REQID : $s.reqID$)
end operation

// Invoked only by the client

operation CS.ARR-RCVRESP($s, clk, resp, reqID$)
  // Verify request identifier
  if $reqID \neq s.reqID$ **then return** s
    // Calc the difference between the server's clock and the client's clock; we add half of the round-trip delay as an approximation
  s.offset $\leftarrow (resp - clk) + (clk - s.ReqTime)/2$
  return s and **ask to invoke** operation CS-INITRESPCLIENT of the **parent** instance
end operation

// Invoked only by the server

operation CS.ARR-RCVREQ($s, clk, req, reqID, ClAddr$)
  return s and **ask to invoke** operation SENDRESP of instance $s.ARRinstance$, with parameters (RESP : clk, REQID : $reqID$), CLADDR : $ClAddr$) *// Send local clock*
end operation

operation CS.GETTIME(s, clk)
  // Return time after correcting it with the offset
  return s and **ask to invoke** operation CS-GETTIMERESP of the **parent** instance, with parameter (TIME:$clk +$ s.offset)
end operation

requests and responses, and a bound on how long it takes to initialize a server or client. In addition, clients which are initialized once should output no more than one initialization response (this is required for our clock synchronization

protocol $\mathcal{P}_{\text{UnauthCS}}$ to work correctly). For convenience, we use the same bound Δ_{ARR} for the two bounds.[19] The authenticated request-response black-box requirement $\mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}}$ includes the four requirements described below.

- **Server authentication ($\mathcal{R}_{\textbf{SRVAUTH}}^{\textbf{ARR}}$, Algorithm 21).** Suppose that (1) ins_S is a benign ARR server instance with address $addr$ and public key PK, (2) ins_C is a benign ARR client instance which communicates with an ARR server with address $addr$ and public key PK, (3) ins_C is assigned[20] address $ClAddr$, and (4) ins_C correctly received response $resp$ and asked to give it to its parent, together with identifier $reqID$. Then:
  1) ins_C sent request req with identifier $reqID$.
  2) ins_S sent response $resp$ with identifier $reqID$ to ins_C.
- **Timely requests and responses ($\mathcal{R}_{\textbf{ARR-REQ-RESP}}^{\Delta_{\text{ARR}}}$, Algorithm 21).** Suppose that (1) ins_S is a benign ARR server instance with address $addr$ and public key PK which completes its initialization at time t_S, (2) ins_C is a benign ARR client instance which communicates with an ARR server with address $addr$ and public key PK and ins_C completes its initialization at time t_C, (3) ins_C is assigned address $ClAddr$, and (4) ins_C correctly received response $resp$ and asked to give it to its parent, together with identifier $reqID$. Then:
  1) **Timely request**: if ins_C sends req with $reqID$, then within Δ_{ARR} time, ins_S correctly receives req and $reqID$ and asks to give req, $reqID$, and $ClAddr$ to its parent.
  2) **Timely response**: if ins_S later sends $resp$ with $reqID$ to ins_C, then within Δ_{ARR} time, ins_C correctly receives $resp$ and $reqID$ and asks to give $reqID$ and $resp$ to its parent.
- **Timely initialization ($\mathcal{R}_{\textbf{ARR-INIT}}^{\Delta_{\text{ARR}}}$, Algorithm 21).** If a benign ARR instance is initialized, then it completes its initialization and invokes its parent within Δ_{ARR} time. The predicate $\mathcal{R}_{sp\text{-INIT}}^{\Delta}.\pi$ is shown in Algorithm 19.
- **One initialization response ($\mathcal{R}_{\textbf{CLIENTINITRESP}}^{\textbf{ARR}}$, Algorithm 21).** Benign client ARR instances, when initialized once, do not output more than one initialization response (by asking to invoke operation ARR-INITRESPCLIENT of the parent instance), which signals completion of initialization.[21]

Using the above requirements, the authenticated request-response requirement $\mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}}$ is defined as:

$$\mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}} = \mathcal{R}_{\text{SRVAUTH}}^{\text{ARR}} \cup \mathcal{R}_{\text{ARR-REQ-RESP}}^{\Delta_{\text{ARR}}} \cup \mathcal{R}_{\text{ARR-INIT}}^{\Delta_{\text{ARR}}}$$

H.1.7. Security Analysis of the $\mathcal{P}_{\text{UnauthCS}}$ Clock Synchronization Protocol. The $\mathcal{P}_{\text{UnauthCS}}$ protocol uses a nonce. The adversary is allowed to see the nonce, i.e., we do not prevent eavesdropping on it. The adversary does not know the nonce in advance, so it cannot get a response with this nonce in advance and give a too-old response to the requesting instance and have the requesting instance accept this response; it can only do this after the requesting instance outputs the nonce. At the same time, the adversary cannot give a future response to the requesting instance, because real time must increase in the execution. Consequently, even if the adversary sees the nonce, it cannot break the freshness property which the nonce ensures.

Theorem 2 ($\mathcal{P}_{\text{UnauthCS}}$ asymptotically satisfies $\mathcal{R}_{\mathsf{CS}}^{\Delta_{\text{CS}}}$ assuming $\mathcal{M}_{\mathsf{CS}}^{\Delta_{\text{CS}}}$ and a blackbox satisfying $\mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}}$)**.** *Let $\Delta_{\text{CS}} \equiv \Delta_{\text{SRVOFF}} + 3\Delta_{\text{ARR}} + 5\Delta_{\text{ASK}}$. The (unauthenticated) clock synchronization protocol $\mathcal{P}_{UnauthCS}$ asymptotically satisfies the clock synchronization requirement $\mathcal{R}_{\mathsf{CS}}^{\Delta_{\text{CS}}}$ assuming model $\mathcal{M}_{\mathsf{CS}}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}$ and a blackbox protocol satisfying requirement $\mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}}$. More precisely:*

$$\mathcal{P}_{UnauthCS} \models_{X^*,\text{POLY}}^{\mathcal{M}_{\mathsf{CS}}^{\Delta_{\text{SRVOFF}},\Delta_{\text{ASK}}}, \mathcal{R}_{\mathsf{ARR}}^{\Delta_{\text{ARR}}}} \mathcal{R}_{\mathsf{CS}}^{\Delta_{\text{CS}}}.$$

Proof:

Let us first consider the CS-OFFSET property. Suppose that the adversary wins, i.e., a CS instance asked to give time value t to its parent at time τ, and $|t - \tau| \geq \Delta_{\text{CS}}$. From the model, the server used by the instance has local time within Δ_{SRVOFF} of the real time, so if the instance is a server, then the claim holds.

. Based on the model, the client was initialized with a CS server. Based on the blackbox requirement, it takes at most Δ_{ARR} time to receive a request or response. Consequently, when the CS server asked to send the time t_S to the client, the t_S was at most Δ_{SRVOFF} off from the real time. The ARR server instance was invoked at most Δ_{ASK} later, then the ARR client instance received the response at most Δ_{ARR} later and asked to give the response containing t_S to its parent, which is the CS client instance. The CS client is invoked at most Δ_{ASK} later, at real time τ'. Thus, t_S was then at least $\tau' - (\Delta_{\text{SRVOFF}} + 2 \cdot \Delta_{\text{ASK}} + \Delta_{\text{ARR}})$ and at most $\tau' + \Delta_{\text{SRVOFF}}$.

At this point, the client calculated and saved the offset as $(t_S - clk) + delay/2$, where $delay$ is the difference in local time measured by the client since it last asked to send a synchronization request, which is at most $2 \cdot \Delta_{\text{ARR}} + 4 \cdot \Delta_{\text{ASK}}$, due to the asking and request and response delays. Based on the model, the client's local clock has a fixed offset from the real time τ, so we can write clk as $\tau + C$, where C is a constant. The offset saved by the client is thus:

19. The two different bounds could be separated. They may have much different values. For example, in the protocol $\mathcal{P}_{\text{ARR}}^{\mathcal{S}}$, the initialization delay would likely be much smaller than the request/response delay. This is because, in $\mathcal{P}_{\text{ARR}}^{\mathcal{S}}$, initialization of an ARR involves initializing a COM instance but does not involve communicating with other entities; in contrast, sending a request or response would likely involve communicating between different entities, which may take much more time.

20. The address is assigned by a subprotocol providing communication, or by the adversary.

21. This is needed in our example, because in the protocol $\mathcal{P}_{\text{UnauthCS}}$, the operation CS.ARR-INITRESPCLIENT chooses a request identifier and sends the request to the CS server. If the CS client's request identifier gets changed due to operation CS.ARR-INITRESPCLIENT being invoked again, then the client will reject the server's response, interfering with timely initialization of the CS subprotocol.

Algorithm 21 The server authentication, timely requests and responses, and one initialization response requirements for ARR protocols ($\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}$, $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR-REQ-RESP}}$, and $\mathcal{R}^{\text{ARR}}_{\text{CLIENTINITRESP}}$, respectively)

Allowed failure probability $\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\beta \equiv 0$
Requirement predicate $\mathcal{R}^{\text{ARR}}_{\text{SRVAUTH}}.\pi(T)$ // ***Server auth.***
    Suppose that (1) instance ins_S of subprotocol ARR, in benign entity ent_S, asked to invoke operation ARR-INITRESPSERVER with parameters ADDR:$addr$ and PK:PK, (2) instance ins_C, running in a benign entity ent_C, has an ARR.INIT event with parameters ROLE:CLIENT, SRVADDR:$addr$ and SRVPK:PK, (3) instance ins_C is assigned address $ClAddr$, and (4) instance ins_C asks to invoke ARR-RCVRESP with parameters RESP:$resp$ and REQID:$reqID$. Then:
        (1) There was an ARR.SENDREQ event in ins_C with parameters REQ; denote the value as req, and REQID; denote the value as $reqID$.
        (2) There was an ARR.SENDRESP event at ins_S, with parameters RESP:$resp$, REQID:$reqID$, and CLADDR:$ClAddr$.
end requirement predicate

Allowed failure probability $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR-REQ-RESP}}.\beta \equiv 0$
Requirement predicate $\mathcal{R}^{\Delta_{\text{ARR}}}_{\text{ARR-REQ-RESP}}.\pi(T)$ // ***Timely R&Rs***
    Suppose that (1) at time t_S, instance ins_S of ARR, in a benign entity, asks to invoke ARR-INITRESPSERVER with parameters ADDR:$addr$ and PK:PK, (2) at time $t_C > t_S$, instance ins_C of (another) benign entity has an ARR.INIT event with parameters ROLE:CLIENT, SRVADDR:$addr$ and SRVPK:PK, (3) at time $t'_C \geq t_C$, instance ins_C asks to invoke ARR-INITRESPCLIENT, and (4) instance ins_C is assigned address $ClAddr$. Then:
        (1) **Timely request**: if at time $t_{SendReq} > t'_C$, there is an ARR.SENDREQ event at ins_C with parameters REQ:req and REQID:$reqID$, then at time t_{RcvReq} such that $t_{SendReq} < t_{RcvReq} \leq t_{SendReq} + \Delta_{\text{ARR}}$, instance ins_S asks for an ARR-RCVREQ event in its parent instance, with parameters REQ:req, REQID:$reqID$, and CLADDR:$ClAddr$.
        (2) **Timely response**: if later, at time $t_{SendResp} > t_{RcvReq}$, there is an ARR.SENDRESP event at ins_S with parameters RESP:$resp$, REQID:$reqID$, and CLADDR:$ClAddr$, then at time $t_{RcvResp}$ such that $t_{SendResp} < t_{RcvResp} \leq t_{SendResp} + \Delta_{\text{ARR}}$, instance ins_C asks for an ARR-RCVRESP event in its parent instance, with parameters RESP:$resp$ and REQID:$reqID$.
end requirement predicate

Allowed failure probability $\mathcal{R}^{\text{ARR}}_{\text{CLIENTINITRESP}}.\beta \equiv 0$
Requirement predicate $\mathcal{R}^{\text{ARR}}_{\text{CLIENTINITRESP}}.\pi(T)$
end requirement predicate

$$
\begin{aligned}
&\mathit{offset} = (t_S - \mathit{clk}) + \mathit{delay}/2 \\
&\leq (t_S - (\tau + C)) + (2 \cdot \Delta_{\text{ARR}} + 4 \cdot \Delta_{\text{ASK}})/2 \\
&= (t_S - \tau) - C + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}
\end{aligned}
$$

where $-(\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) \leq (t_S - \tau) \leq \Delta_{\text{SRVOFF}}$. Combining the above, we derive two inequalities:

$$
\begin{aligned}
&\mathit{offset} = (t_S - \mathit{clk}) + \mathit{delay}/2 \\
&\leq (t_S - \tau) - C + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}} \\
&\leq \Delta_{\text{SRVOFF}} - C + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}} \\
&= (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) - C
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{offset} = (t_S - \mathit{clk}) + \mathit{delay}/2 \\
&\geq t_S - \mathit{clk} \\
&= t_S - (\tau + C) \\
&= (t_S - \tau) - C \\
&\geq -(\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) - C
\end{aligned}
$$

In $\mathcal{P}_{\text{UnauthCS}}$, if the instance is initialized, then CS.GETTIME returns the time immediately when invoked. It calculates the time value t as $t = \mathit{clk} + \mathit{offset}$, where offset is the offset previously saved during synchronization, which we analyzed above. From the model, the client's local clock offset from real time is fixed. Consequently, when CS.GETTIME is invoked at the CS client at real time τ'', when its local clock time is clk'', then the CS client immediately returns $t'' = \mathit{clk}'' + \mathit{offset}$, which is bounded as follows, based on the above inequalities:

$$
\begin{aligned}
&t'' = \mathit{clk}'' + \mathit{offset} \\
&= \tau'' + C + \mathit{offset} \\
&\leq \tau'' + C + (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) - C \\
&= \tau'' + (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}})
\end{aligned}
$$

and

$$\begin{aligned} & t'' = clk'' + offset \\ & = \tau'' + C + offset \\ & \geq \tau'' + C + -(\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) - C \\ & = \tau'' - (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}) \end{aligned}$$

Thus, we get that the time value t'' returned by the CS client at real time τ'' is at least $\tau'' - (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}})$ and at most $\tau'' + (\Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}})$. It follows that $|t'' - \tau''| \leq \Delta_{\text{SRVOFF}} + \Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}$, which is $\leq \Delta_{\text{CS}}$. This implies that $|t - \tau'| \leq \Delta_{\text{CS}}$, which is a contradiction.

Let us next consider the CS-DELAY property. Suppose that the adversary wins, i.e., the execution ended after real time $\tau + \Delta_{\text{CS}}$, and one of three cases happened: (1) operation CS.INIT was invoked at time τ with role 'server', but by time $\tau + \Delta_{\text{CS}}$, the CS server instance did not ask to invoke CS-INITRESPSERVER at its parent, (2) operation CS.INIT was invoked at time τ with role 'client', but by time $\tau + \Delta_{\text{CS}}$, the CS client instance did not ask to invoke CS-INITRESPCLIENT at its parent, or (3) operation CS.GETTIME was invoked at time τ, which was after the instance was initialized, but by time $\tau + \Delta_{\text{CS}}$, the CS instance did not ask to invoke CS-GETTIMERESP at its parent.

In the case of CS.GETTIME, in $\mathcal{P}_{\text{UnauthCS}}$, CS.GETTIME returns the time, asking to invoke CS-GETTIMERESP at its parent, immediately if the instance was initialized. Consequently, the delay would be 0, not $> \Delta_{\text{CS}}$.

In the case of CS.INIT with role = 'server', in $\mathcal{P}_{\text{UnauthCS}}$, the CS server asks to initialize a new ARR server instance. Based on the model, it takes at most Δ_{ASK} to invoke the ARR server instance (1st Δ_{ASK}). Based on the blackbox requirement, it takes at most Δ_{ARR} for the ARR server to complete its initialization and ask to invoke ARR-INITRESPSERVER at its parent, which is the CS server (1st Δ_{ARR}). Then, again based on the model, at most Δ_{ASK} later, the CS server is invoked with operation CS.ARR-INITRESPSERVER (2nd Δ_{ASK}). The CS server immediately asks to invoke CS-INITRESPSERVER at its parent. Thus, it takes at most $\Delta_{\text{ARR}} + 2 \cdot \Delta_{\text{ASK}}$ from when CS.INIT is invoked with role = 'server' to when the instance asks to invoke CS-INITRESPSERVER at its parent. This delay is $\leq \Delta_{\text{CS}}$.

Finally, in the case of CS.INIT with role = 'client', in $\mathcal{P}_{\text{UnauthCS}}$, the CS client asks to initialize a new ARR client instance. Based on the model, it takes at most Δ_{ASK} to invoke the ARR client instance (1st Δ_{ASK}). Based on the blackbox requirement, it takes at most Δ_{ARR} for the ARR client to complete its initialization and ask to invoke ARR-INITRESPCLIENT at its parent, which is the CS client (1st Δ_{ARR}). Then, again based on the model, at most Δ_{ASK} later, the CS client is invoked with operation CS.ARR-INITRESPCLIENT (2nd Δ_{ASK}).

At this point, the CS client asks send a synchronization request to the CS server, using the ARR client instance. After at most Δ_{ASK}, the ARR client is invoked (3rd Δ_{ASK}). Based on the blackbox requirement, it takes at most Δ_{ARR} until the ARR instance used by the CS server receives the request and asks to give it to operation ARR-RCVREQ at its parent, which is the CS server (2nd Δ_{ARR}). Then, in at most Δ_{ASK}, CS.ARR-RCVREQ is invoked to receive the request at the CS server (4th Δ_{ASK}). The CS server immediately asks to send a response containing a time value to the CS client, using the ARR server instance. After at most Δ_{ASK}, the ARR server is invoked (5th Δ_{ASK}). Based on the blackbox requirement, it takes at most Δ_{ARR} until the ARR instance used by the CS client receives the request and asks to give it to operation ARR-RCVRESP at its parent, which is the CS client (3rd Δ_{ARR}). The CS client verifies the request identifier it receives with the response, calculates and saves its offset, and asks to invoke CS-INITRESPCLIENT at its parent.

Thus, it takes at most $3 \cdot \Delta_{\text{ARR}} + 5 \cdot \Delta_{\text{ASK}}$ from when CS.INIT is invoked with role = 'client' to when the instance asks to invoke CS-INITRESPCLIENT at its parent. This delay is $\leq \Delta_{\text{CS}}$.

This contradicts the assumption that one of the three cases which allow the adversary to win happened.

□

H.2. Authenticated Request-Response Protocol Implementation and Security Analysis

H.2.1. The $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ Authenticated Request-Response Protocol. The protocol $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ (Algorithm 22) implements the ARR scheme using a standard signature scheme $\mathcal{S} = \{\text{KEYGEN}, \text{SIGN}, \text{VER}\}$. Our implementation is based on asymmetric cryptography, i.e., it uses private and public keys for authentication. This could be changed to symmetric cryptography if the assumed model was adjusted accordingly. Note that we do not treat the signature scheme as a subprotocol, with its own separate instances and states, because it includes only stateless functions, which can be run directly by the operations of $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$. In addition, the implementation relies on a communication subprotocol COM that handles the delivery of packets, with two operations: COM $= \{\text{INIT}, \text{SENDPKT}\}$, following the discussion in §G.

At initialization, i.e., in ARR.INIT, a client instance stores the server's address and public key, as provided from the parent protocol, and completes the initialization by asking to invoke the parent's ARR-INITRESPCLIENT operation. For server instances, ARR.INIT generates an asymmetric key pair and asks to invoke the COM.INIT operation, which will provide the server its network addressing information. COM.INIT will provide this information by invoking the ARR.COM-INITRESP operation. When that happens, the server instance stores the address and invokes the parent's ARR-INITRESPSERVER operation, providing both the server's address and public key.

In ARR.SENDREQ, the client instance receives a request and a request identifier. This identifier allows to map a response to a previously-sent request, as the response will have the same identifier. Then, ARR.SENDREQ asks to invoke the COM.SENDPKT operation (to handle the actual packet delivery), providing the request identifier and the request itself. (The COM instance or the adversary is expected to

Algorithm 22 The $\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ authenticated request-response protocol

operation ARR.INIT($s, clk, role, SrvAddr, SrvPK$)
  if $s.role \neq \perp$ **then return** s // *Instance was already initialized, so ignore*
  $s.role \leftarrow role$
  if $role =$ 'client' **then**
    $(s.SrvAddr, s.SrvPK) \leftarrow (SrvAddr, SrvPK)$
  else
    $s.role \leftarrow$ 'server'
    $(s.SK, s.PK) \leftarrow \mathcal{S}.\text{KeyGen}(s.1^\kappa)$ // *Generate key pair for signing and verifying signatures*
  end if
  return s and **ask to initialize** a (new) instance of subprotocol COM, identified as $s.COMinstance$
end operation

operation ARR.COM-INITRESP($s, clk, addr$)
  $s.addr \leftarrow addr$
  if $role =$ 'client' **then**
    return s and **ask to invoke** operation ARR-INITRESPCLIENT of the **parent** instance
  end if
  return s and **ask to invoke** operation ARR-INITRESPSERVER of the **parent** instance, with parameters (ADDR:$s.addr$, PK:$s.PK$)
end operation

operation ARR.SENDREQ($s, clk, req, reqID$) // *Invoked at client to send a request*
  if $s.role \neq$ 'client' **then return** $s, (\text{'Error='}, \top)$
  return s and **ask to invoke** operation SENDPKT of instance $s.COMinstance$, with parameters (PAYLOAD:(REQID:$reqID$, REQ:req), DSTADDR:$s.SrvAddr$)
    // Model forces the adversary to provide also the client's address
    in the field SRCADDR
end operation

// Get as input two important fields of the packet from the COM layer: the payload and the source address
operation ARR.COM-RCVPKT($s, clk, payload, srcAddr$)
  if $s.role =$'client' **then**
    $m \leftarrow payload.reqID||payload.resp$
    if $\mathcal{S}.\text{Verify}_{s.SrvPK}(m, payload.\sigma) \neq \top$ **then return** $s, (\text{'Error='}, \top)$
    return s and **ask to invoke** operation ARR-RCVRESP of the **parent** instance, with parameters (RESP:$payload.resp$, REQID:$payload.reqID$)
  else
    return s and **ask to invoke** operation ARR-RCVREQ of the **parent** instance, with parameters (REQ:$payload.req$, REQID:$payload.reqID$, CLADDR:$ClAddr$)
  end if
end operation

operation ARR.SENDRESP($s, clk, resp, reqID, ClAddr$) // *Invoked at server to send a response*
  if $s.role \neq$ 'server' **then return** $s, (\text{'Error='}, \top)$
  return s and **ask to invoke** operation SENDPKT of instance $s.COMinstance$, with parameters (PAYLOAD:(RESP:$resp$, REQID:$reqID$, σ:$\mathcal{S}.\text{Sign}_{s.SK}(reqID||resp)$), (ADDR:$ClAddr$))
end operation

provide the source address, i.e., the address of the client, with the payload.) In ARR.SENDRESP, the request identifier and the response are signed by the server's private key and sent back to the client via the COM.SENDPKT operation.

The requests are received by the server in the ARR.COM-RCVPKT operation; more precisely, the COM subprotocol receives the packet and invokes the ARR.COM-RCVPKT operation with the received payload,

which includes the request, the request identifier, and in the case of a response, a signature, and the source address, which is not part of the payload. Since ARR does not authenticate client messages, the server instance simply invokes the parent's ARR-RCVREQ operation with the request, the request identifier, and the source address as the client address. However, when client instances invoke ARR.COM-RCVPKT, they first verify the signature with the server's public key. If valid, the response and the request identifier are delivered to the parent's ARR-RCVRESP operation.

H.2.2. Security Analysis of the $\mathcal{P}^{\mathcal{S}}_{\mathbf{ARR}}$ Authenticated Request-Response Protocol.

Theorem 3 ($\mathcal{P}^{\mathcal{S}}_{\text{ARR}}$ asymptotically satisfies $\mathcal{R}^{\Delta_{\text{ARR}}}_{\mathsf{ARR}}$ assuming $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$)**.** *Let $\Delta_{\text{ARR}} \equiv \Delta_{com}$. Then $\mathcal{P}^{\mathcal{S}}_{ARR}$ asymptotically satisfies the authenticated request-response requirement $\mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}$, assuming the $\mathcal{M}^{\Delta_{com}}_{COM}$ [∞] model, and that $\mathcal{S}$ is a secure signature scheme. Namely: for any secure signature scheme $\mathcal{S}$:*

$$\mathcal{P}^{\mathcal{S}}_{ARR} \models^{\mathcal{M}^{\Delta_{com}}_{COM}[\infty]}_{\overline{X}^*,\text{POLY}} \mathcal{R}^{\Delta_{\text{ARR}}}_{ARR}.$$

Proof sketch: the *server authentication* property follows from the security of the signature scheme $\mathcal{S}$; the *timely requests and responses* property and the *timely initialization* property follow from the $\mathcal{M}^{\Delta_{com}}_{\text{COM}}$ model. □