AUC: Accountable Universal Composability

Mike Graf, Ralf Küsters, and Daniel Rausch
University of Stuttgart
Stuttgart, Germany
Email: {mike.graf,ralf.kuesters,daniel.rausch}@sec.uni-stuttgart.de

Abstract

Accountability is a well-established and widely used security concept that allows for obtaining undeniable cryptographic proof of misbehavior, thereby incentivizing honest behavior. There already exist several general purpose accountability frameworks for formal game-based security analyses. Unfortunately, such game-based frameworks do not support modular security analyses, which is an important tool to handle the complexity of modern protocols.

Universal composability (UC) models provide native support for modular analyses, including re-use and composition of security results. So far, accountability has mainly been modeled and analyzed in UC models for the special case of MPC protocols, with a general purpose accountability framework for UC still missing. That is, a framework that among others supports arbitrary protocols, a wide range of accountability properties, handling and mixing of accountable and non-accountable security properties, and modular analysis of accountable protocols.

To close this gap, we propose *AUC*, the first general purpose accountability framework for UC models, which supports all of the above, based on several new concepts. We exemplify AUC in three case studies not covered by existing works. In particular, AUC unifies existing UC accountability approaches within a single framework.

1. Introduction

Accountability is a prominent concept that is widely used in security. Many security properties and applications, such as auctions [7, 29, 97], e-voting [1, 65, 67], non-repudiation [98–100], multi-party computation (MPC) [6, 8, 13, 54], public key infrastructures (PKIs) [61, 73, 76], distributed ledgers [15, 16, 38, 40, 47, 50, 59, 82, 88], DRM [91, 93], power infrastructures [58, 75], content delivery networks [31], and distributed systems [52, 90, 94] make use of and rely on accountability to provide security. In the formal security analysis literature, so-called *property*-, *policy*-, or *goal-based* accountability is the standard and most commonly used interpretation of accountability (e.g., [33, 42–45, 52, 57, 62, 63, 67, 68, 79]), which we therefore also consider in this work and often just call *accountability* (see also Section 2.7). Property-based accountability intuitively states that, if some intended security property of a protocol, such as correctness of the output, is violated, then we obtain a cryptographic proof that one or more protocol participants have misbehaved. We call security properties ensured via accountability in this way *accountability(-based) properties*. Parties which violated an accountability property can be held accountable for their misbehavior, e. g., via contractual and financial penalties or by excluding them from future protocol runs. This serves as a strong incentive for malicious parties to honestly follow the protocol and to not break security goals.

Over the past decade, researchers have developed general tools and approaches to formally analyze accountability properties in game-based settings (e. g., [46, 62, 67]). These works cover many flavors of accountability, such as *(i)* different *accountability levels*; a weak level might guarantee identification only of a (potentially large) group such that at least one party in that group has misbehaved, where the strongest level, so-called *individual accountability* [67], allows for identifying one or more parties such that *all* of them have misbehaved. *(ii) Local* or *public/universal* accountability (w.r.t. some security goal/property) [41, 49]; a public accountability property allows everyone, including external observers, to identify misbehaving parties in case the security property is broken. Local accountability allows only protocol participants to identify misbehaving parties.

Preventive vs. Accountability Properties. The counterpart to accountability properties are so-called *preventive security properties* (cf. [45]), which includes many special cases such as *proactive security* [2, 12]. In preventive security, one proves that the expected security property cannot be broken in the first place, typically based on certain (strong) security assumptions. In contrast, accountability-based security accepts that security properties might be broken by misbehaving parties but instead requires that one can identify and hence deter such parties. In exchange for this weaker security guarantee, accountability provides several advantages, including: *(i)* accountability might already be achievable with simpler, more efficient components. *(ii)* In order to ensure accountability properties, one might need less security assumptions. *(iii)* Accountability properties might be stronger in some aspects than preventive security notions. For example, Graf et al. [50] illustrate and leverage all of these advantages of accountability by proposing a slight modification of the Hyperledger Fabric permissioned blockchain that *(i)* uses only a very efficient crash-fault-tolerant instead of a more complex Byzantine-fault tolerant consensus protocol, *(ii)* does not assume an honest (super)majority or any other set of honest parties to achieve accountability of consistency, and *(iii)* can enforce consistency (in an accountable way) not only for honest nodes but also for dishonest ones. So both concepts, preventive and accountability-based security, come with their own merits and tradeoffs. They are *orthogonal* concepts in the sense that both types of security can be used as stand-alone mechanisms to provide security for an intended property. But they can also be combined for the same security goal, where accountability serves as a second layer of defense in case the underlying assumptions of preventive security are broken. Such a combination is, for instance, used by the system PeerReview [52] to strengthen the security property of *consistency* in Byzantine-fault tolerant (BFT) consensus protocols (e. g., [28, 83, 95]): as long as there is an honest supermajority in the BFT protocol, consistency cannot be broken at all. If this assumption is no longer met, then PeerReview running on top of the BFT protocol still provides accountability w.r.t. consistency, i. e., allows for identifying parties that cause the consensus protocol to fail.

Universal Composability. Universal composability (UC) (e. g., [17, 18, 22, 23, 53, 64]) is a very popular approach for designing, modeling, and analyzing security protocols due to its strong security guarantees and its inherent support for modular design and analysis. Roughly speaking, in UC one first specifies an *ideal protocol* (or *ideal functionality*) $\mathcal{F}$ that specifies the intended behavior/security properties of a target protocol, abstracting away implementation details. For a concrete realization – the *real protocol* – $\mathcal{P}$, one then proves that "$\mathcal{P}$ behaves just as $\mathcal{F}$" in arbitrary contexts, where the network of $\mathcal{F}$ is controlled by a benign attacker called simulator. One can then build other protocols $\mathcal{P}'$ on top of $\mathcal{F}$ and analyze their security. A so-called composition theorem provided by the underlying UC model then implies that $\mathcal{P}'$ remains secure even if the subroutine $\mathcal{F}$ is replaced by $\mathcal{P}$.

Current State. Preventive security properties and their formalization via ideal functionalities in UC models is a well-studied problem, with the vast majority of existing UC literature focusing on preventive properties. In contrast, accountability properties have only been studied and formalized for special cases, mostly MPC protocols (e. g., [13, 14, 24, 34, 35, 55, 74]). These works do not and were not intended to serve as general UC accountability frameworks. In particular, these works *(i)* are tailored towards MPC protocols and accountability properties thereof, such as *identifiable abort* (e. g., [14, 55], cf. Appendix G, where we capture identifiable within our formalization approach), *(ii)* assume certain protocol structures, such as non-interactive protocols or protocols without internal servers, *(iii)* typically focus on either local or public accountability, *(iv)* cannot express arbitrary relationships of properties, including preventive properties with accountability as second layer of defense, and/or *(v)* consider composition out of scope or focus on composition of certain protocol types.

Hence, a general framework for accountability in UC models is still missing. Such a general framework would

1. Enable the design and analysis of arbitrary protocols and accountability properties within a UC model, thereby allowing such protocols to benefit from strong security statements and modularity offered by UC models,
2. Provide a common tool set that allows for easily formalizing accountability properties and reducing the effort for protocol designers, and thus,
3. Help avoid mistakes and oversights that can otherwise easily occur when formalizing accountability for every use case from scratch, as well as
4. Facilitate the comparison of different types of accountability properties and guarantees as they are defined within the same overarching and unified framework.

Obtaining such a general framework for accountability in UC models is non-trivial as it has to support and combine a number of different features to achieve the desired generality. Among others, it has to be able to express many different flavors of accountability, including the following flavors from formal game-based accountability security frameworks [62, 63, 67, 79]: *(i)* accountability for a wide range of security properties, *(ii)* different levels of accountability, *(iii)* both local and public accountability properties, and *(iv)* relationships of different properties, including combinations of accountability and preventive properties both as independent concurrent properties but also as layered defense for the same security goal. Such a general framework further must be able to express virtually arbitrary protocols. In particular, it must *(v)* be independent of a specific protocol type and structure, thereby supporting, e. g., interactive protocols and protocols with purely internal parties such as client-server protocols, and *(vi)* fully support the modular design, analysis, and composition of protocols with accountability.

AUC – Accountable Universal Composability. To close this gap, we propose the *Accountable Universal Composability (AUC) framework*, the first general framework for the modular UC analysis of accountability properties. The AUC framework works within existing UC models, such as the UC [22, 23], including its variants (e.g., SUC [24], GUC [18]), the GNUC [53], the CC [77], and the IITM [64] models. By this, AUC inherits and can leverage features of the underlying UC model, such as the respective composition theorems possibly including support for composition with joint, arbitrarily shared, and/or global state. This also allows protocol designers to compose AUC protocols with existing (preventive security) protocols while remaining within whatever model they are already familiar with.

A major component of AUC is a generic transformation that, given any ideal functionality, allows for incorporating a wide range of accountability properties into the functionality. AUC further enables modeling and analyzing accountability properties in the corresponding real protocols such that composition of the resulting (accountability-based) protocols, also with preventive security protocols, is fully supported. To this end, AUC generalizes several ideas from the literature, both in game-based and UC settings, but also adds novel concepts, such as what we call *judicial reports* and *supervisors*. By combining these concepts, AUC achieves all the previously mentioned goals and features of a general framework for property-based accountability.

To exemplify features, applications, and the generality of AUC, we present the first accountability analyses of three different case studies in a UC model. These case studies are chosen to be relatively simple to better illustrate AUC. Firstly, we show how an accountable consensus service can be scaled up, e. g., to support a larger number of clients, by adding a scaling protocol layer on top while retaining accountability of the overall protocol. Using composability, this result can be iterated arbitrarily often to obtain security of multiple scaling layers. The case study introduces and illustrates general techniques that can be used for future analyses of accountability of existing real-world protocols that follow a similar scaling approach, such as the prominent blockchain Hyperledger Fabric [3, 50], the consensus service Hashgraph [11], and content delivery networks [39]. Secondly, we model and analyze accountability of a public key infrastructure (PKI) based on certificate transparency logs (CTLs). We then, thirdly, compose this result with an ISO 9798-3-based authenticated key exchange [17, 26, 56, 66], showing that the resulting protocol provides security based on an accountable PKI. To the best of our knowledge, this analysis is the first UC analysis of that protocol without assuming pre-distributed public keys or an idealized PKI where the adversary cannot create certificates for honest parties.

Contributions. In summary, the contributions of the paper are as follows:

- We propose AUC, the first general framework for accountability in UC models.
- AUC transfers and generalizes existing concepts from game-based approaches to the UC world, generalizes existing UC approaches, and develops new concepts.
- AUC supports, among others, arbitrary ideal and real protocols, a wide range of accountability properties, local and public accountability properties, concurrent consideration and combination of preventive and accountability properties, accountability of protocol internal parties, and composition of accountable protocols.
- To exemplify AUC, we present three case studies, providing the first UC analysis of accountability properties for the considered protocols.
- As a sanity check, in Appendix G we further show that AUC can capture accountability aspects of existing UC MPC literature as a special case, thereby generalizing and unifying this line of work.

Structure of this paper. Section 2 presents AUC, including a discussion of its core concepts. Section 3 provides our case studies. We discuss related work in Section 4. Further details are given in the appendix.

2. AUC – Accountable Universal Composability

In this section, we introduce the accountable universal composability (AUC) framework. We first clarify notation and terminology in Section 2.1. Section 2.2 discusses several high-level ideas of AUC before we formally specify the AUC transformation for ideal functionalities in Section 2.3, with AUC's modeling of real protocols covered in Section 2.4. In Section 2.5, we discuss several aspects of AUC's composability abilities. In Section 2.6, we present a deterrence analysis to analyze the behavior of rational adversaries. Section 2.7 concludes this section with a discussion on AUC.

2.1 Notation and Terminology

Computational Model. Formally, we define AUC within the iUC model [17], an easy to use but fully expressive instantiation of the IITM model [64]. However, AUC and its concepts can also be used within arbitrary other models for universal composability, e. g., [23, 53, 77]. We keep our presentation on a level such that readers familiar with these UC models can understand and use AUC without requiring any prior knowledge of iUC. For interested readers, we provide a brief introduction to the iUC model including an overview of its pseudo code notation in Appendix A and in Appendix B, we provide an overview of the used pseudo code notation.

A party in iUC and hence also AUC is uniquely identified via its party ID pid, the session sid it runs in, and the piece of code/role $role$ it is executing. We call the triple $(pid, sid, role)$ *entity*.[1] In what follows, we use the terms entity and party synonymously.

We call a party in a protocol *main* if it can directly receive inputs from and send outputs to the environment. We call a party *internal* otherwise, i. e., if it is part of an internal subroutine. Whether a party is an internal or a main party can be determined from its role. As in all UC models, an ideal functionality and a realization share the same sets of main parties/roles. A realization might have additional internal parties/roles, such as an internal server used by main clients, that are not present in the ideal protocol.

As is standard in UC models, the adversary $\mathcal{A}$ is allowed to corrupt parties by sending a special corrupt command. If an entity is corrupted, the adversary generally gets full control over the (input and output interface of the) entity. The environment can obtain the current corruption status of main parties in a protocol, which allows for checking whether corruption of main parties is simulated correctly.

Classes of Security Properties. As mentioned in the introduction, the literature traditionally distinguishes between *preventive security properties* and *accountability properties*. We denote the set of accountability properties of a protocol by $\mathcal{S}ec^{\text{acc}}$ and divide preventive properties into two classes:

Absolute: A preventive security property is called *absolute* if all underlying assumptions used in the security proof are assumed to always hold true. The analysis of the case where such assumptions might become broken is out of scope. We denote the set of absolute security properties (of some protocol) by $\mathcal{S}ec^{\text{abs}}$.

Assumption-based: We call a preventive security property *assumption-based* if it is shown to hold true under certain assumptions but the security analysis also analyzes the case that these assumptions might become broken at some point. We denote the set of assumption-based preventive security properties by $\mathcal{S}ec^{\text{assumption}}$.

By this categorization, we have that absolute security properties can neither be assumption-based properties nor accountability properties, i. e., we require $\mathcal{S}ec^{\text{abs}} \cap (\mathcal{S}ec^{\text{assumption}} \cup \mathcal{S}ec^{\text{acc}}) = \emptyset$. The set $\mathcal{S}ec^{\text{assumption}} \cap \mathcal{S}ec^{\text{acc}}$, if non-empty, contains preventive security properties that offer accountability as a second layer of defense whenever the underlying assumptions are broken. The set $\mathcal{S}ec^{\text{assumption}} \setminus \mathcal{S}ec^{\text{acc}}$ contains those assumption-based security properties which are not additionally secured via accountability.

Verdicts. AUC defines verdicts to be positive boolean formulas consisting of propositions of the form $\mathsf{dis}(A_i)$ where A_i is an entity and $\mathsf{dis}(A_i)$ expresses that the judge believes that A_i misbehaved/is dishonest. For example, the verdict "$\mathsf{dis}(A_1) \wedge (\mathsf{dis}(A_2) \vee \mathsf{dis}(A_3))$" captures the statement that party A_1 and at least one of the parties A_2 and A_3 have misbehaved. We can evaluate verdicts by setting $\mathsf{dis}(A_i) = \texttt{true}$ iff A_i is actually a corrupted entity at the point where the verdict is stated, and `false` otherwise. We call a verdict *fair* if it evaluates to `true`, and hence, does not mistakenly blame honest parties. In a secure protocol, all honestly computed verdicts are required to be fair.

This definition of verdicts allows for capturing different levels of accountability. Verdicts such as $\mathsf{dis}(A_1)$ or $\mathsf{dis}(A_1) \wedge \mathsf{dis}(A_2)$ imply that the specific party A_1 (and potentially others) misbehaved. This captures the strongest level of so-called *individual accountability*. In contrast, a verdict of the form $\mathsf{dis}(A_1) \vee \mathsf{dis}(A_2) \vee \mathsf{dis}(A_3)$ only identifies a group of three parties where at least one has misbehaved, therefore capturing a weaker level of accountability.

2.2 Overview of AUC's Central Concepts

The AUC framework serves as a general blueprint for modeling and analyzing a wide range of accountability properties both in real *and* ideal protocols. Before delving into AUC, let us first give a high-level overview of its main concepts and ideas:

[1]In those UC models that identify parties only via the pair (pid, sid), different roles can be modeled by adding them as a prefix to pid, say, $pid = (role, pid')$, where pid' is the actual party ID.

Breaking accountability properties in exchange for verdicts: In an (AUC) accountable ideal functionality, the adversary $\mathcal{A}$ may, at any point in time, instruct the functionality to break/disable accountability properties. In exchange for breaking an accountability property, the ideal functionality requires $\mathcal{A}$ to provide a verdict that indicates parties who are blamed for the security breach. This verdict must be *fair*, i. e., it may not blame parties who were honestly following the protocol. Verdicts are received and their fairness is checked by so-called (ideal) *judges*. A judge is a new role added to the ideal functionality that models parties who are responsible for determining misbehaving protocol participants. The environment and higher-level protocols, including higher-level judges, can ask for a judge's verdict.
In a realization of the ideal functionality, the corresponding (real) judges specify the exact judging procedures, such as checking signatures, and the exact input data used as evidence to compute verdicts. For example, in an e-voting protocol that is supposed to provide accountability w.r.t. counting votes (a strong form of so-called *end-to-end verifiability* [33]), a real judge – run by an auditor or even a voter – might take as input all messages from a bulletin board and then blame parties who produce output, e. g., the election result, but with invalid accompanying zero-knowledge proofs.

Judge Types: AUC considers an a priori unbounded number of concurrent judges, which can be instantiated by protocol designers to capture different numbers and types of judges executed by different parties modeling various flavors of accountability. Most common are *public* and *local judges*, which capture *public* and *local accountability*, respectively: *(i)* A single public judge implements public accountability, i. e., verdicts of the (real) public judge are computed solely based on publicly available information. For example, in e-voting to check that the tallying process went correctly, (cf. "accountability w.r.t. counting votes" above) a public judge typically uses data, such as zero-knowledge proofs, from a public bulletin board. Since the data used is public, everyone, including outside observers, can take the role of a public judge. It therefore typically makes sense to model a public judge as an incorruptible entity. *(ii)* Several local judges, each of them belonging to and typically representing the validation procedure executed by one protocol participant, model local accountability. That is, verdicts of a (real) local judge are computed by a protocol participant, say Alice, and can therefore be based not just on public information and whatever data other parties are willing to provide to Alice, but verdicts can also be based on Alice's own private data, possibly allowing for detecting a wider variety of misbehavior. For example, in e-voting Alice, resp. her local judge, can tell that her ballot does not appear on the bulletin board, even though she submitted it. Since a local judge is run by a (potentially malicious) protocol participant who might lie about the verdict, a local judge is typically considered corrupted iff the corresponding protocol participant is corrupted. *(iii)* Additionally, AUC also supports other types of judges, such as, *mandated judges*. A mandated judge models a (potentially trusted) external auditor which (in the realization) computes verdicts based on evidence that protocol participants provide. For instance, in some e-voting protocols aiming for everlasting privacy [36] not all data needed for verifiability of the election can be published. Instead, only a trusted auditor would obtain all necessary data, including potentially non-public data, to perform the verification procedure.
We emphasize that in AUC protocol designers are free to define judges in whatever way suitable. In particular, protocol designers can decide what kind of information judges use/require, whether the information is public or private, what groups of parties they are responsible for, or whether judges are corruptible. Hence, AUC does not dictate specific types of judges, and as mentioned, also does not bound the number of judges considered.

Judge dependent accountability properties: We take the view that a judge is responsible for one or more security properties and one or more parties (not necessarily exclusively). That is, if a security properties is broken from the point of view of a party, then a judge responsible for this security property and party is supposed to output a verdict; conversely, without such a verdict the property should still hold true for this party. For example, consider again accountability w.r.t. counting votes in e-voting. Here it makes sense to have a public judge responsible for all parties and the correctness of the tallying process, who would output a verdict when the tallying process was carried out incorrectly. To give another example, in Section 3.2 we consider a PKI based on Certificate Transparency Logs (CTL). Here, a security property is that there should not be certificates on the CTL containing a public key that does not belong to the alleged party, say Alice. This property may be broken for Alice but not other parties, and only Alice can check this property. So here we would have a local judge responsible for Alice carrying out the necessary checks; as long as this judge does not output a verdict, all of Alice's certificates on the CTL should be correct.
We formalize the above in AUC by allowing the adversary in the ideal protocol to mark an accountability

property p as broken for a specific set of judges, and require these judges to output verdicts. From then on, the property p is no longer enforced by the ideal functionality for those parties that are protected by that set of judges.

Judicial reports: Judges can provide arbitrary information, such as an aggregated view of collected evidence to higher-level protocols via the novel concept of *judicial reports*. Judges in such higher-level protocols can then in turn also use this information for computing their own verdicts. This is crucial to enable modular design and analysis of a wide range of accountability-based protocols.

Assumption-based properties: Similarly to accountability properties, AUC formalizes assumption-based properties in ideal protocols by allowing the adversary to mark the underlying assumptions as broken. If that property is not additionally protected by accountability, then the property itself is also considered broken. In general, the assumptions and corresponding properties might hold true for some parties but not for others. For example, Alice might achieve *liveness* since all of her messages were delivered within a *bounded network delay* but Bob might no longer have liveness as some of his messages were dropped. Hence, just as for judge dependent accountability properties, the ideal functionality allows the adversary to mark assumptions as broken for a specific set of affected parties. Those parties then lose security guarantees, unless they are still protected by accountability.

Supervisor: A new meta party called *supervisor* collects information about *(i)* corruption of internal protocol participants (i. e., those that exist only within subroutines of the real protocol but not in the ideal protocol) and *(ii)* broken security assumptions. This information is provided by the supervisor to the environment to guarantee that the real and ideal worlds coincide in these aspects. In other words, a simulator cannot cheat but has to keep these aspects consistent.

2.3 AUC Transformation for Ideal Functionalities

We now explain how AUC turns an arbitrary ideal functionality into an accountable one. Let $\mathcal{F}$ be an arbitrary (non-accountable) ideal functionality that enforces a set of preventive security properties $\mathcal{S}ec$, e. g., correctness, consistency, or liveness. AUC provides a general transformation $\mathcal{T}(\cdot)$ for such ideal functionalities $\mathcal{F}$ that creates an accountable ideal functionality $\mathcal{F}^{\text{acc}}$ where arbitrary subsets of the properties from $\mathcal{S}ec$ are changed to instead be assumption-based and/or accountability properties.[2] That is, $\mathcal{F}^{\text{acc}}$ ensures a combination of absolute properties $\mathcal{S}ec^{\text{abs}}$, assumption-based properties $\mathcal{S}ec^{\text{assumption}}$, and accountability properties $\mathcal{S}ec^{\text{acc}}$. The transformation $\mathcal{T}(\cdot) := \mathcal{T}_2(\mathcal{T}_1(\cdot))$ consists of two steps $\mathcal{T}_1$ and $\mathcal{T}_2$ that progressively modify $\mathcal{F}$ to obtain $\mathcal{F}^{\text{acc}} = \mathcal{T}(\mathcal{F})$. The first step adds the necessary infrastructure to $\mathcal{F}$ to be able to express accountability properties, such as code for the new judge and supervisor roles, but does not yet alter the actual behavior and security guarantees of $\mathcal{F}$. The second step $\mathcal{T}_2(\cdot)$ then changes the behavior of $\mathcal{F}$ to model the effects of broken security properties. Next, we define each step.

Step 1: The transformation $\mathcal{T}_1(\mathcal{F}) =: \mathcal{F}'$ takes as input the original ideal functionality to create an intermediate functionality $\mathcal{F}'$. This step adds a fixed set of parameters, variables, and static code to $\mathcal{F}$ that defines the set of accountability properties, judges, a supervisor, verdicts, as well as related operations for the adversary (cf. Figures 1 to 3). The parameters are designed to be instantiated by a protocol designer to customize aspects of $\mathcal{F}^{\text{acc}}$ that depend on the specific real protocols at hand, e. g., the exact accountability level one wants to analyze.

The full version of $\mathcal{T}_1(\cdot)$ builds the entire range of AUC features into $\mathcal{F}'$. This should be seen as a general blueprint. Features not needed in the application at hand can be omitted.

Structural changes. Figure 1 specifies structural components, parameters, and state variables that are added by $\mathcal{T}_1$ to $\mathcal{F}$. More specifically, the roles of judge and supervisor are added to $\mathcal{F}$, as well as their corruption behavior: The supervisor is purely a modeling tool, hence incorruptible. As explained in Section 2.2, whether judges are corruptible mainly depends on the types of judges that are modeled and is therefore not a priori fixed by AUC. Protocol designers rather flexibly specify the corruption model for judges by instantiating the newly added subroutine $\mathcal{F}_{\text{judgeParams}}$ (this new subroutine is also used to customize further aspects such as

[2]It is straightforward to also add entirely new assumption-based and/or accountability properties to $\mathcal{F}$ while applying the AUC transformation. For simplicity of presentation we will leave this option implicit.

Additional roles: judge, supervisor
Additional protocol parameters: *{They may be polynomially checkable predicates*

- $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$ *{Accountability properties*
- $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$ *{Assumption-based security properties*
- $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$ *{set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)*
- $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$ *{set of entities/IDs where properties are ensured via assumptions*

Additional subroutines: $\mathcal{F}_{\mathrm{judgeParams}}$
Additional Corruption behavior:

```
– AllowCorruption($pid, sid, role$):
    Do not allow corruption of ($pid, sid$, supervisor).
    if $role$ = judge:
        send (Corrupt, ($pid, sid$, judge), internalState)
          to ($pid, sid, \mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)      {$\mathcal{F}_{\mathrm{judgeParams}}$ decides whether judges can be corrupted
        wait for $b$
        return $b$
– DetermineCorrStatus[a]($pid, sid, role$):
    if $role$ = judge:                                            {$\mathcal{F}_{\mathrm{judgeParams}}$ may determine a judge's corruption status
        send (CorruptionStatus?, ($pid, sid$, judge), internalState)
          to ($pid, sid, \mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)
        wait for $b$; return $b$
– AllowAdvMessage($pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m$)
    Do not allow sending messages to $\mathcal{F}_{\mathrm{judgeParams}}$.   {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}_{\mathrm{judgeParams}}$ in the name of corrupted parties.
```

Additional internal state:

- $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\mathtt{true}, \mathtt{false}\}$ *{Stores broken security properties per judge/id, initially* false *∀entries*
- $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ *{Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially ε*
- $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\mathtt{true}, \mathtt{false}\}$ *{Stores broken security assumptions per id, initially* false *∀entries*
- $\mathsf{corruptedIntParties} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{b} \cup \{\mathtt{judge}, \mathtt{supervisor}\})$, initially $\emptyset$ *{The set of corrupted internal parties $(pid, sid, role)$*

[a]**DetermineCorrStatus** allows protocol designers to specify whether an entity that is currently not directly controlled by the attacker should nevertheless consider itself to be corrupted. E.g., a local judge will typically consider itself to be corrupted already if its corresponding party is corrupted.

[b]$\mathsf{Roles}_{\mathcal{F}}$ is the set of (main) roles provided by $\mathcal{F}$ to the environment. For example, $\mathsf{Roles}_{\mathcal{F}} = \{\mathtt{signer}, \mathtt{verifier}\}$ for an ideal signature functionality $\mathcal{F} := \mathcal{F}_{\mathrm{sig}}$.

Fig. 1: Parameters and state added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

accountability levels; we explain this later). For example, for local judges $\mathcal{F}_{\mathrm{judgeParams}}$ would typically disallow corruption if the corresponding protocol participant is honest, and conversely consider the local judge to be corrupted as soon as the protocol participant is corrupted. As mentioned, the adversary gains full control over corrupted judges. Further, AUC adds the parameters $\mathsf{Sec}^{\mathsf{acc}}$ and $\mathsf{Sec}^{\mathsf{assumption}}$ to $\mathcal{F}$, which are instantiated by the protocol designer to contain exactly those accountability and assumption-based properties ($\subseteq \mathcal{S}ec$) she wants to consider.

The parameter $\mathsf{pids}_{\mathsf{judge}}$ specifies the considered (possibly infinite) set of party IDs of (different types of) judges. For example, the most common types of judges mentioned in Section 2.2 can be model via PIDs $pid_j \in \mathsf{pids}_{\mathsf{judge}}$ of the following form: *(i)* $pid_j = \mathtt{public}$ identifies a unique public judge, *(ii)* $pid_j = (\mathtt{local}, pid, role)$ models a local judge of protocol participant $(pid, sid, role)$, and *(iii)* $pid_j = (\mathtt{mandated}, pid)$ highlights a mandated judge with PID pid.

In the state variable brokenProps we track for each combination of a security property $\in \mathsf{Sec}^{\mathsf{acc}}$ and judge $\in \mathsf{pids}_{\mathsf{judge}}$ whether the property is broken for that judge (and hence, not guaranteed anymore for the parties this judge is responsible for). Jumping slightly ahead, the second transformation step $\mathcal{T}_2(\cdot)$ will change the behavior of the functionality depending on which properties are marked broken for which judges and their governed parties. The map verdicts stores the current verdict of each judge.

The parameter $\mathsf{ids}_{\mathsf{assumption}}$ defines the (possibly infinite) set of IDs for which an assumption and the corresponding assumption based property might become broken. For example, *(i)* $id = \mathtt{public}$ can be used to model a global assumption and property that affects all parties, whereas *(ii)* $id = (\mathtt{local}, pid, role)$ can model an assumption and corresponding property specific to a protocol participant $(pid, sid, role)$. For each combination of property $\in \mathsf{Sec}^{\mathsf{assumption}}$ and ID $\in \mathsf{ids}_{\mathsf{assumption}}$, we track whether underlying assumptions are currently broken in the new variable brokenAssumptions. Once assumptions are broken, then the property itself might also become broken for the affected ID (see below), which again is tracked in brokenProps.

Finally, we add the set corruptedIntParties which tracks corrupted internal parties in a realization in addition to corrupted main parties that are already tracked by the functionality.

```
Additional code for the judge role:
    recv ($\texttt{BreakAccProp}, verdict, toBreak$) from NET[a] to ($pid, sid, \texttt{judge}$)
        s.t. $toBreak \subseteq \mathsf{Sec}^{\mathsf{acc}} \times \mathsf{pids}_{\mathsf{judge}} \wedge verdict$ maps from $\mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$:
        $(successful, leakage) \leftarrow \texttt{breakAttempt}(verdict, toBreak)$          {breakAttempt is defined below
        reply ($\texttt{BreakAccProp}, successful, leakage$)
    recv GetVerdict from I/O to ($pid_j, sid, \texttt{judge}$):          {The environment can query the verdicts of local and public judges
        reply ($\texttt{GetVerdict}, \mathsf{verdicts}[pid_j]$)
    recv ($\texttt{GetJudicialReport}, msg$) from I/O to ($pid_j, sid, \texttt{judge}$):   {The environment may query for local or public judicial reports
        send ($\texttt{GetJudicialReport}, msg, \mathsf{internalState}$) to ($pid_j, sid, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams}$)   {Forward judicial report request to $\mathcal{F}_{\mathrm{judgeParams}}$
        wait for ($\texttt{GetJudicialReport}, report$)
        reply ($\texttt{GetJudicialReport}, report$)
Helperfunctions:
    procedure $\texttt{breakAttempt}(verdict, toBreak)$ :          {Process break attempt
        Check for all non-$\varepsilon$ verdicts in $verdict$, i.e., $\forall pid_j$ s.t. $verdict[pid_j] \neq \varepsilon$:
          1. it holds true that $verdict[pid_j]$ is a positive boolean expression built from propositions of the form $\mathsf{dis}((pid, sid, role))$,
          2. it holds true that $\mathsf{eval}(verdict[pid_j]) = \texttt{true}$,[b]
        Check that $\forall (prop, pid_j) \in toBreak$:
          3. $verdict[pid_j] \neq \varepsilon$,
          4. $\mathsf{brokenAssumptions}[prop, pid_j] = \texttt{true}$, if $prop \in \mathsf{Sec}^{\mathsf{assumption}} \wedge pid_j \in \mathcal{S}ec^{\mathsf{assumption}}$.
        if any of the above check fails:
            return($\texttt{false}, \varepsilon$)
        send ($\texttt{BreakAccProp}, verdict, toBreak, \mathsf{internalState}$) to ($\_, \_, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams}$)
        wait for ($\texttt{BreakAccProp}, successful, leakage$)
            {$\mathcal{F}_{\mathrm{judgeParams}}$ can impose further conditions, e.g., on verdicts or whether it is allowed to violate breakable security properties.
        if $successful$:
            for all $\forall pid_j$ s.t. $verdict[pid_j] \neq \varepsilon$ do:
                $\mathsf{verdicts}[pid_j] \leftarrow verdict[pid_j]$          {Record accepted local and public verdict
            for all $(prop, pid_j) \in toBreak$ do:
                $\mathsf{brokenProps}[prop, pid_j] \leftarrow \texttt{true}$
        return($successful, leakage$)
```

[a]NET denotes message from the network adversary. I/O denotes messages from the environment.

[b]eval evaluates the bolean expression, where $\mathsf{dis}(pid, sid, role)$ evaluates to true if $(pid, sid, role) \in$ CorruptionSet or $(pid, sid, role) \in$ corruptedIntParties. CorruptionSet is a predefined variable of iUC that contains all corrupted main parties of this functionality. We set $\mathsf{eval}(\varepsilon) :=$ true.

Fig. 2: Judge code added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

```
Additional code for the supervisor role:                                {$\mathcal{A}$ may break these assumptions
    recv ($\texttt{BreakAssumption}, toBreak$) from NET to ($\_, \_, \texttt{supervisor}$) s.t. $toBreak \subseteq \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}}$:
        for all $(prop, id) \in toBreak$ do:
            $\mathsf{brokenAssumptions}[prop, id] \leftarrow \texttt{true}$          {Record broken assumptions
            if $prop \notin \mathsf{Sec}^{\mathsf{acc}} \vee id \notin \mathsf{pids}_{\mathsf{judge}}$:
                $\mathsf{brokenProps}[prop, id] \leftarrow \texttt{true}$
                    {Record property as broken if not additionaly secured via accountability
        send ($\texttt{BreakAssumption}, toBreak, \mathsf{internalState}$)
          to ($pid, sid, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams}$)
                    {$\mathcal{F}_{\mathrm{judgeParams}}$ provides leakage
        wait for ($\texttt{BreakAssumption}, leakage$)
        reply ($\texttt{BreakAssumption}, leakage$)
    recv ($\texttt{corruptInt}, (pid, sid, role)$) from NET to ($\_, \_, \texttt{supervisor}$)
        s.t. $role \notin \mathsf{Roles}_{\mathcal{F}} \cup \{\texttt{judge}, \texttt{supervisor}\}$:     {$\mathcal{A}$ is allowed to corrupt internal protocol parties
        $\mathsf{corruptedIntParties.add}((pid, sid, role))$
        reply ($\texttt{corruptInt}, \texttt{ack}$)
    recv ($\texttt{IsAssumptionBroken?}, prop, id$) from I/O
      to ($\_, \_, \texttt{supervisor}$) s.t. $id \in \mathsf{ids}_{\mathsf{assumption}}$:     {The environment may ask whether properties are broken
        if $prop \in \mathsf{Sec}^{\mathsf{assumption}}$:
            reply ($\texttt{IsAssumptionBroken?}, \mathsf{brokenAssumptions}[prop, id]$)
        else:
            reply ($\texttt{IsAssumptionBroken?}, \perp$)
    recv ($\texttt{corruptInt?}, (pid, sid, role)$) from I/O to ($\_, \_, \texttt{supervisor}$)
        s.t. $role \notin \mathsf{Roles}_{\mathcal{F}} \cup \{\texttt{judge}, \texttt{supervisor}\}$:
                    {The environment may ask for the corruption status of internal parties
        if $(pid, sid, role) \in \mathsf{corruptedIntParties}$:
            reply ($\texttt{corruptInt}, \texttt{true}$)
        else:
            reply ($\texttt{corruptInt}, \texttt{false}$)
```

Fig. 3: Supervisor code added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

Judges. Figure 2 specifies the code AUC adds to ideal functionalities to model judges. Each judge in AUC is an entity of the form $(pid_j, sid, \texttt{judge})$ where pid_j is the judge's PID running the fixed `judge` role.

The adversary can try to break accountability properties $\in \mathsf{Sec}^{\mathsf{acc}}$ by sending a `BreakAccProp` message to the ideal functionality, which contains a list of properties including the judges $\subset \mathsf{pids}_{\mathsf{judge}}$ for which these properties shall be broken as well as a list of new verdicts for (some of the) judges. After receiving the message, the ideal functionality first checks whether this attempt meets all minimal requirements for accountability, i. e., *(i)* all (non-empty) verdicts are positive boolean formulas, *(ii)* all verdicts are fair based on the current corruption status of main and internal parties, *(iii)* if a property is marked as broken for some judge, then that judge also outputs a verdict, and *(iv)* accountability properties that are also assumption-based properties may not be broken as long as the underlying assumptions still hold true. If the attempt passes these checks, $\mathcal{F}'$ forwards the attempt (including its full internal state) to the subroutine $\mathcal{F}_{\mathrm{judgeParams}}$, which decides whether the attempt actually succeeds and whether/which information is leaked to the attacker, say, because a privacy property was broken. By instantiating $\mathcal{F}_{\mathrm{judgeParams}}$ a protocol designer can therefore customize the exact level of accountability and also relationships between properties. For example, an instance of $\mathcal{F}_{\mathrm{judgeParams}}$ might require verdicts to have the form "$\mathsf{dis}(A)$", i.e., identify exactly one misbehaving party, thereby providing individual accountability (cf. Section 3 for examples with different accountability levels). It might also require that, if a property connected to a party is broken, then the same property must also be broken concurrently for others, capturing the relationship that several/all parties are affected and thus able to compute verdicts simultaneously (cf. Appendix G).

The environment can query judges of $\mathcal{F}'$ to obtain their current verdicts and their judicial reports. For verdicts, $\mathcal{F}'$ returns the last accepted verdict. For reports, $\mathcal{F}'$ calls $\mathcal{F}_{\mathrm{judgeParams}}$ which can compute the report based on the entire internal state of $\mathcal{F}'$. This allows a protocol designer to customize which information is contained in such reports by instantiating $\mathcal{F}_{\mathrm{judgeParams}}$ appropriately (see Section 3.1 for an example).

Supervisor. Figure 3 presents the code added for the supervisor. The adversary can send a `BreakAssumption` message to mark the assumptions as broken that underlie some properties $p \in \mathsf{Sec}^{\mathsf{assumption}}$ for some set of IDs $\subset \mathsf{ids}_{\mathsf{assumption}}$. Assumption-based properties also protected by accountability, i.e., $p \in \mathsf{Sec}^{\mathsf{acc}}$ and $id \in \mathsf{pids}_{\mathsf{judge}} \cap \mathsf{ids}_{\mathsf{assumption}}$, are not yet marked as broken; for these, the adversary still has to issue a `BreakAccProp` message to id with a valid verdict. Otherwise, breaking the underlying assumption also breaks the property. The adversary may further mark arbitrary internal parties as corrupted. These parties can then also be blamed in verdicts.

As mentioned, the environment can ask the supervisor whether assumptions are marked as broken and whether internal parties are marked as corrupted. Note that $\mathcal{F}'$ does not impose any limitations on when assumptions can be marked as broken but only ensures – by providing this information to the environment – that this occurs if and only if assumptions are broken in the realization. By this, the realization can specify the exact conditions and limitations for broken assumptions without requiring any modifications to $\mathcal{F}'$ each time a different realization is considered (cf. Sections 2.4 and 2.7).

Step 2: The second step of the transformation $\mathcal{T}_2$ specifies the effects of a broken property. Observe that the exact implications in terms of behavior of $\mathcal{F}$ strongly depend on the individual security properties. Therefore, $\mathcal{T}_2$, unlike $\mathcal{T}_1$, cannot simply be a fixed set of variables, code that need to be added to a functionality, or even a black-box transformation of $\mathcal{F}'$. Instead, $\mathcal{T}_2$ rather constitutes a more abstract guideline on how the functionality $\mathcal{F}'$ has to be modified. Importantly, such modifications must not alter the behavior when security guarantees hold true since we want to retain the same security guarantees of $\mathcal{F}$ in those cases:

Definition 1. *Let $\mathcal{F}' := \mathcal{T}_1(\mathcal{F})$. Let $\mathcal{F}^{acc} := \mathcal{T}_2(\mathcal{F}')$ be a functionality obtained by introducing additional behavior for capturing broken security properties. We say that $\mathcal{F}^{acc}$ is an* accountable transformation *of $\mathcal{F}$ if the behavior of $\mathcal{F}^{acc}$ and $\mathcal{F}'$ is identical in all runs until a security property is marked as broken.*

Technically, modeling the effects of a broken security property $p \in \mathsf{Sec}^{\mathsf{acc}} \cup \mathsf{Sec}^{\mathsf{assumption}}$ affecting some ID $id \in \mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}$ (specifying, e.g., an affected party), generally entails introducing (one or more) conditional clauses of the form "**if** (p, id) *is not marked as broken* **then** *<original behavior>* **else** *<new behavior>*". As the name suggests, *<original behavior>* denotes the original unchanged behavior of the functionality $\mathcal{F}$, i.e., the code that enforces p. The code *<new behavior>* then defines what "breaking p" actually means, typically by giving more power to the adversary.

For example, if $\mathcal{F} := \mathcal{F}_{\mathrm{sig}}$ is an ideal signature functionality and $p = \texttt{unforgeability}$ (for, say, $id = \texttt{public}$, modeling public unforgeability), then *<original behavior>* is a check during signature validation

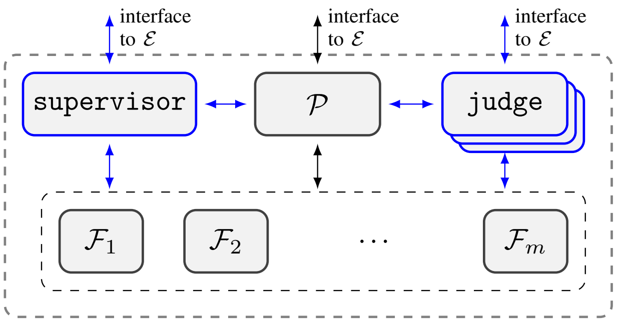


Fig. 4: Example of machines and connections in an accountable real protocol $\mathcal{P}$ with ideal (possibly accountable) subroutines $\mathcal{F}_1, \ldots, \mathcal{F}_m$. Blue components are added by AUC.

that, if it detects forgery of a signature, returns `false` irrespective of the actual result of signature validation. Breaking this property would simply disable this forgery check, i.e., *<new behavior>* is empty. Thus, if a signature is forged if `unforgeability` is broken, the transformed version of $\mathcal{F}_{\text{sig}}$ might actually return `true`.

More complex conditional statements can be used as well to capture advanced relationships of properties. For example, consider `privacy` in MPC protocols. The statement "**if** *there are at least t PIDs pid_i of local judges such that* $(\texttt{privacy}, pid_i)$ *is marked as broken* **then** *<leak secret information>*" captures a threshold relationship of local properties: in order to break privacy, the adversary has to mark privacy as broken for at least t parties.

Observe that all transformations $\mathcal{T}_2$ following the form described above always satisfy Definition 1. That is, the resulting ideal functionalities $\mathcal{F}^{\text{acc}} := \mathcal{T}_2(\mathcal{T}_1(\mathcal{F}))$ are indeed accountable transformations of $\mathcal{F}$ as per Definition 1.

2.4 AUC in Real Protocols

As illustrated in Figure 4, modeling accountability in a real protocol $\mathcal{P}$ using AUC mainly entails adding and specifying a supervisor and several judges that correspond to the ones in the ideal protocol. Again, not always all of these components are needed and can be omitted if not used.

Judges. The concrete definitions of judges in real protocols formalize *(i)* the judging algorithms used for computing verdicts, *(ii)* inputs and hence evidence needed for obtaining the verdict, *(iii)* which parties are supposed to provide which information as evidence, and *(iv)* to whom evidence is provided, namely, the party that is running the judge. We exemplify modeling of real judges for the most common types of public, local, and mandated judges: Intuitively, *a public judge* computes verdicts solely based on publicly available/verifiable data such as information from a public bulletin board or data that individual parties are willing to provide and therefore publish even to external observers. Since an honest party can always locally execute the public judge, the public judge is modeled to be incorruptible. To make formally explicit in the security analysis that also an attacker might run the public judge, public judges should generally publish all collected information via their network interface to the environment. For the same reason, if a public judge is allowed to interact with other parts of the protocol, e.g., to verify a signature in an ideal signature functionality $\mathcal{F}_{\text{sig}}$, then typically the judge should provide a network interface for the environment to be able to perform the same interactions.[3] We provide a concrete example of a public judge in Section 3.1. *Local judges*, typically one per (main) protocol party, use public information and also (private) data that protocol participants are willing to share with the party running the local judge but might not want to fully publish. In addition, since a local judge is run by a party herself, it also takes as input the entire internal state and possibly history of that corresponding party, including any private information, to compute a verdict. A local judge should be considered to be corrupted iff its corresponding protocol participant is corrupted. We provide concrete examples of local judges in Sections 3.2 and 3.3. *Mandated judges* reflect the judging procedure of an external mandated auditor, which takes as input publicly available data and also (private) data that protocol participants are willing to share with the auditor. Whether (some of the) mandated judges are corruptible depends on the setting that shall be modeled.

[3]This modeling of public judges is similar in spirit to modeling random oracles. A random oracle represents a public function, which is captured by being incorruptible but providing outputs via an additional network interface to the environment.

When evaluating evidence, real judges often obtain or compute additional information that might be useful for and can even be required by higher-level protocols. Such information can be shared via judicial reports (we discuss the need for this novel concept, including example use cases, in Section 2.7).

Supervisor. A supervisor in a real protocol forwards the corruption status of internal protocol participants and specifies when exactly the assumptions underlying an assumption-based property for some party/ID must be considered broken.

This generally involves gathering data from other parts of the protocol. For example, consider a property $p \in \mathcal{Sec}^{\text{assumption}}$ that relies on an *honest majority* assumption and thus affects everyone (i.e., $id = \texttt{public}$), e. g., consistency in Bitcoin [48]. That is, a protocol only ensures p if more than half of the protocol participants are honest. To determine whether the protocol still ensures p, the supervisor would check whether a majority of the protocol participants is still honest. If the majority is lost, the supervisor would indicate that the assumptions for p are no longer met and thus the protocol might no longer guarantee p.

As we discuss in Section 2.7, our novel concept of a supervisor is necessary to be able to capture a wide range of assumption-based properties as well as real protocols with arbitrary internal structures, such as client-server protocols (e. g., [14, 55]). Our case studies in Sections 3.1 and 3.2 provide concrete examples and indeed require a supervisor.

2.5 Composable Security Analysis in AUC

A security analysis in AUC consists of a realization proof where one shows that the protocol $\mathcal{P}^{\text{acc}}$ at hand indeed realizes (in the UC sense) the accountable ideal functionality $\mathcal{F}^{\text{acc}}$. Among others, this formally proves that $\mathcal{P}^{\text{acc}}$ enjoys the desired security properties, including accountability properties. Since AUC works within existing UC models and uses the standard realization notion, one can now, as usual, build an (accountable or traditional) higher-level protocol $\mathcal{Q}^{\text{acc}}$ using $\mathcal{F}^{\text{acc}}$ as a subroutine (written $(\mathcal{Q}^{\text{acc}} \mid \mathcal{F}^{\text{acc}})$) and prove that it is secure, i.e., realizes some $\mathcal{F'}^{\text{acc}}$. The composition theorem of the underlying UC model then immediately implies that also the composed protocol $(\mathcal{Q}^{\text{acc}} \mid \mathcal{P}^{\text{acc}})$ using $\mathcal{P}^{\text{acc}}$ instead of $\mathcal{F}^{\text{acc}}$ still realizes $\mathcal{F'}^{\text{acc}}$, i.e., achieves all desired strict, assumption-based, and/or accountability properties.

In an accountable higher-level protocol $\mathcal{Q}^{\text{acc}}$, the judges can and typically will obtain verdicts and judicial reports from the subroutine judges in $\mathcal{F}^{\text{acc}}$ resp. $\mathcal{P}^{\text{acc}}$. This information can then be used by higher-level judges to compute their own verdicts and reports (cf. the BFT example in Section 2.7 and our case studies in Sections 3.1 and 3.3 for higher-level judges that rely on lower level judges). We note that we do not impose any restrictions on which lower-level judges a higher-level judge may access. For example, often a higher-level local judge of some party will only use local subroutine judges belonging to the same party. This models that the party executes all of its judges from all protocol layers iteratively and can re-use the results of previous computations, such as verdicts identifying misbehavior in subroutines, in the following computations. However, a higher-level local judge can also, e.g., obtain the verdicts of lower level judges of different parties, modeling that one party uses the (claimed) results of other parties. While these results might not necessarily be trustworthy, a higher-level judge might, e.g., be able to aggregate and perform majority voting to obtain a fair result, or achieve a weaker level of accountability that considers the possibility of the subroutine judge lying in their verdict.

2.6 Deterrence Analysis

In addition to the UC security analysis, protocol designers should perform a cost-benefit/deterrence analysis of the (possibly composed) real protocol to determine whether honest parties are willing to take part in the protocol and whether accountability indeed deters rational adversaries from misbehavior. This analysis can be performed using any standard approach from the literature. As a simple example, based on the approach by Asharov et al. [6] we describe one possible concrete mechanism for this purpose. For a rational acting honest party p_i, one considers the utility/profit U^i_{hP} for running the protocol, the cost U^i_{hE} for disclosing private data as evidence to judges, and the loss due to (falsely) accusation U^i_{hL} (by a corrupted judge). For a rational but maliciously acting party p_i, U^i_{mP} is the potential utility/profit from misbehaving, U^i_{mL} is the cost for misbehaving (e.g., reputation loss or contractual penalties after being detected), and U^i_{mE} is the cost for providing (possibly maliciously crafted) evidence to judges. Now, accountability provides a suitable security mechanism in a practical deployment if

$$U^i_{hP} - U^i_{hE} - U^i_{hL} \geq 0 \text{ and} \tag{1}$$
$$U^i_{hP} - U^i_{hE} - U^i_{hL} \geq U^i_{mP} - U^i_{mE} - U^i_{mL}, \tag{2}$$

i.e., honest parties benefit from and hence are willing to take part in the protocol (Eq. (1)) and malicious parties are deterred from misbehaving as they stand to gain more when honestly following the protocol (Eq. (2)).

Often, the result of such an analysis will be obvious. For example, if CAs in a PKI are detected misbehaving, they typically have to close business [92], i.e., U^i_{mL} will be much higher than U^i_{mP}. We also note that utilities/costs might depend on the context/higher-level protocol that a subroutine will be deployed in. For example, the potential profit U^i_{mP} for tempering with an accountable bulletin board is very large if it is used as subroutine in an e-voting protocol for a major political election. But U^i_{mP} might be negligible if the bulletin board is just one out of several redundant backups in a distributed cloud storage protocol. In such cases a deterrence analysis cannot be performed for individual components but should rather be performed after fully composing the entire real protocol.

2.7 Discussion

AUC is the first general purpose accountability framework for UC models. It builds on and extends existing concepts but also introduces entirely new concepts that are required to construct such a general framework. Here we discuss these concepts and relate them to existing literature.

Accountability properties: AUC formalizes a property-based interpretation of accountability, i. e., security properties might be broken as long as misbehaving parties can be identified and blamed. This is a standard interpretation that is widely established and used in the area of formal protocol security analyses [33, 42–45, 52, 57, 62, 63, 67, 68, 79]. There are other (often informal) interpretations of accountability, also outside of the field security (cf. [46]). A relatively close one in the domain of security requires that any object/message/action can be connected to its originator, e. g., by requiring all parties to sign (cf., e. g., [25, 27]). This interpretation is orthogonal to the property-based interpretation. For example, in e-voting it is important that one cannot trace back ballots to individual voters but one can still achieve property-based accountability for such protocols, namely correctness/verifiability of the election result (see, e. g., [69]). Conversely, even if the election servers sign all their messages, it might not be possible to verify from those messages whether all votes were counted correctly. That is, the protocol might not provide accountability for correctness of the election result.

Verdicts: The game-based model of Küsters et al. [67] defines verdicts as boolean formulas to be able to express different levels of accountability. We transfer this concept to the setting of universal composability. This allows AUC to capture a wide range of common accountability levels, from strong individual accountability to very weak levels where, say, a verdict only says that someone misbehaved but gives no further information on who exactly is at fault.

Judges, Relationships, and Judge Dependent Properties: Judges are an integral part of the specification of protocols since they define the routines, including inputs, that are used by that protocol to detect misbehavior. A formal analysis can then verify for a given detection routine whether it provides (property-based) accountability. Hence, judges are used (at least implicitly) by all works that formally analyze (property-based) accountability (e. g., [6, 13, 34, 37, 54, 62, 63, 67, 79, 87]).
There is a division in the existing literature concerning relationships of security properties: On the one hand, prior general frameworks for accountability such as [62, 63, 67, 79] consider only a single (public/local) judge running at the same time. Hence, these general frameworks cannot actually capture relationships of properties where different parties and/or different types of judges are affected at the same time. This includes, e.g., all local judges obtaining the same verdict or a security property holding true until at least a threshold of t judges (possibly a mixture of local, mandated, and/or a public one) detect misbehavior. On the other hand, there are works that have already considered and analyzed such relationships by proposing specialized security models that hard code a certain relationship for a specific property and setting. This includes, for example, [6, 8, 14, 55, 85] which require for *identifiable abort* in MPC protocols that honest parties agree on the culprit, i.e., their local judges compute *the same* verdict.

AUC is the first general accountability framework that can capture such relations. At its core, this is enabled by considering *multiple concurrent judges and judge types* as well as *judge dependent properties*. Appendix G illustrates this feature.

Judicial reports: This novel concept introduced by AUC allows judges from different protocol layers to exchange information, which is often required to perform a modular analysis.
For example, consider a publicly accountable BFT consensus algorithm used as a subroutine by a higher-level protocol $\mathcal{P}^{\text{acc}}$. As long as the (lower-level) public judge J_{BFT} in the BFT algorithm does not detect misbehavior, she will typically be able to compute the consensus established among the parties from the available evidence. In $\mathcal{P}^{\text{acc}}$, misbehaving parties might be able to break security properties by deviating from the consensus established in the BFT subroutine. Hence, for a higher-level public judge $J_{\mathcal{P}^{\text{acc}}}$ to detect who deviated from the consensus he must first learn the correct consensus. But $J_{\mathcal{P}^{\text{acc}}}$ cannot compute this on his own since, by UC composition, he does not see the internals of the BFT subroutine but only gets restricted access via a limited interface. AUC solves this issue and enables a modular analysis by extending this interface via judicial reports: J_{BFT}, who has full access within the BFT subroutine and who can thus compute the unique consensus if it exists, can provide this information as part of a report to $J_{\mathcal{P}^{\text{acc}}}$. See also our case study in Section 3.1 for a composed protocol whose modular analysis requires judicial reports.

Supervisor: AUC is the first framework that allows for modeling and analyzing the combination of assumption-based and accountability-based security for the same properties in a UC model. It is also the first framework that supports modeling and analyzing accountability of real protocols with arbitrary internal parties without assuming any specific internal structure. Both of these features are enabled by, among others, the novel concept of a supervisor.
To formalize the guarantees of assumption-based properties the behavior of the ideal functionality has to change depending on whether the assumption currently holds true. Note that for many assumptions an ideal functionality cannot actually determine whether they are still met as they often depend on internals that only exist in the realization. For example, liveness properties of blockchains typically assume that an internal network subroutine in the real protocol has a bounded message delivery delay; such a network subroutine does not necessarily exist in the ideal functionality. The supervisor solves this issue: firstly, the realization of the supervisor can specify the exact conditions under which an assumption holds true while having full access to all information of the real protocol. By allowing the environment to query whether assumptions are currently broken, the supervisor then further ensures that the assumption will be marked as broken in the ideal functionality if and only if the conditions specified in the real world are no longer met. Hence, e. g., an ideal blockchain functionality would enforce liveness iff the real blockchain supervisor determines that the real network still has a bounded message delay. Our case study in Section 3.1 underlines the need for this aspect of supervisors.
Verdicts in (possibly composed) real protocols might have to blame internal protocol parties, e. g., servers in client-server protocols [50, 85], where clients are main parties that are available to higher-level protocols while servers are purely internal subroutines. Since only main parties but not any of the internal parties of the real protocol also exist in the corresponding ideal protocol, we use the supervisor to ensure that the simulator can mark internal parties as corrupted in the ideal functionality iff they are corrupted in the real protocol (the same property is already guaranteed for the main parties by the underlying model). This mechanism is necessary to ensure that verdicts which are fair in the ideal functionality are also fair in the realization. Our case studies in Sections 3.1 to 3.3 rely on this use of the supervisor.

Composition: The combination of the above concepts is what allows AUC, for the first time, to capture UC compositions of arbitrary accountability-based protocols. This is further illustrated by our case studies in Sections 3.1 and 3.3, which are the first modular accountability analyzes for these settings and protocols.
Since AUC works within an arbitrary UC model, AUC inherits and preserves all features of the underlying composition theorems. In particular, if the underlying model supports composition with global state (e. g., [9, 17, 18, 71]), then it is possible to make some or all of the subroutine judges within a composed protocol globally available to the environment and arbitrary other concurrent protocols.[4] Whether it is sensible to consider globally available or rather private subroutine judges depends, as with most other global functionalities, on the protocol and the context that is modeled. For example, local judges within composed MPC protocols (cf. Appendix G)

[4]We note that, as observed by [9, 17, 71], the same UC security proof of the subroutine already implies composition with and without globally available subroutine judges.

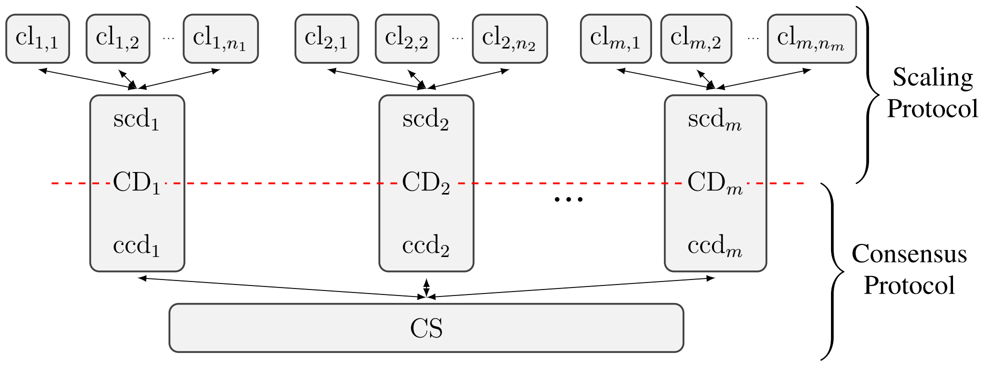


Fig. 5: Consensus scaling: a central service CS establishes consensus. Consensus is distributed via distributors $\mathrm{CD}_1, \ldots, \mathrm{CD}_m$. The CD connect to CS via their CS-clients $\mathrm{ccd}_1, \ldots, \mathrm{ccd}_m$. The CDs distribute consensus via server components $\mathrm{scd}_1, \ldots, \mathrm{scd}_m$ to their clients $\mathrm{cl}_{1,1}, \mathrm{cl}_{1,2}, \ldots$

should typically be modeled as private: the subroutine is supposed to be used only within a single context, namely the MPC protocol, and there is no reason for Alice to share the verdicts and judicial reports of her internal subroutines with arbitrary other parties, protocols, and the environment. In contrast, one might consider modeling judges belonging to a PKI subroutine (cf. Section 3.2) to be globally available within a composed protocol if the same PKI is supposed to be shared by multiple different protocols.
We note that the main difference between global and private subroutine judges is whether only one or multiple higher-level protocols can be composed with the same subroutine. Hence, when only a single specific higher-level protocol has to be considered, then an analysis with a private subroutine is generally already sufficient.

3. Case Studies

In this section, we exemplify the usage and features of AUC via three case studies: an accountable scaling protocol, a simplified version of the Web PKI including CTLs, and a key exchange protocol based on an accountable PKI. We provide for each case study a brief deterrence analysis in the appendix. Additionally – as a sanity check – we cast existing accountability definitions from the UC MPC literature into AUC in Appendix G.

3.1 Scaling Accountable Consensus

In this case study, we analyze a common situation from practice, namely, a scaling protocol built on top of a consensus service (CS), cf. Figure 5. The purpose of this scaling protocol is to increase throughput and hence support a larger number of clients. This is achieved by introducing an additional layer of intermediate servers (called CD in Figure 5) that regularly obtain the established ordered sequence of messages/the consensus from the underlying CS, cache the result, and then use this cache to answer incoming requests from clients. This scaling approach is commonly used, e.g., by the prominent Hyperledger Fabric blockchain [3, 50], the Hashgraph consensus service [11], and content delivery networks [39].

More specifically, in this case study our goal is to scale a *consensus service* that provides *public and individual accountability*, takes inputs from clients, appends them to a globally unique ordered list/state, and gives all clients access to this global state. In particular, if a client does not obtain a prefix of the (same) global state, then a judge, based on public evidence, can blame a misbehaving party. We want to show that the composition of a suitable scaling protocol on top of this service retains all properties of the underlying consensus service, most notably public individual accountability. Note that consensus might fail due to misbehavior of parties in CS but could also be introduced by the scaling layer.

This case study exemplifies several features and aspects of the AUC framework, including *(i)* accountability and assumption-based security properties, *(ii)* public accountability, *(iii)* individual accountability, *(iv)* composition using judicial reports, which are required to express this case study, and *(v)* complex protocol structures, namely a client-server protocol with *internal*, potentially malicious parties/servers. Full details, including formal specifications of all UC machines, functionalities, and a full security proof are available in Appendix D.

The scaling protocol. We consider a scaling protocol that has the structure depicted in Figure 5 and works as described in what follows. While this scaling protocol is a custom one that we chose to illustrate AUC, the general construction is similar to Hyperledger Fabric and Hashgraph. Hence, this case study can serve as a basis for the first UC accountability analysis of these protocols in future work.

Transaction submission: When a client, say $\mathrm{cl}_{1,1}$, submits a transaction (after receiving that transaction from some higher-level protocol), she adds her identity, signs the resulting transaction, and then sends (via an unprotected network) the signed transaction to her *consensus distributor* (CD), here CD_1. The CD is divided into two components: a server part called SCD which interacts with higher-level clients and a client part CCD which interacts with the CS. An SCD verifies the signature of incoming transactions from higher-level clients and then starts acting as a client CCD towards the CS to add the signed transaction to the globally ordered state. This involves running the code of a client, e.g. ccd_1, as specified by the underlying consensus protocol. Depending on the consensus protocol, this client code might, e. g., also add its own signature to the signed transaction.

Accessing the global state: A SCD, say scd_1, regularly calls the client code ccd_1 to obtain a current copy of the global state from the CS. scd_1 then signs (with the key of CD_1) and caches this global state. Whenever a client, such as $\mathrm{cl}_{1,1}$, queries its SCD for the global state, the SCD, here scd_1, responds by returning the most recent signed cached copy of the global state, hence, reducing the load of the CS. $\mathrm{cl}_{1,1}$ accepts and outputs the message to a higher-level protocol if the signature by CD_1 on the whole state is valid.

Security properties. We consider two security properties for both the underlying and the scaled consensus protocols, formalized by an ideal functionality (see below):

Public individual accountability w.r.t. consistency: Clients (both of the scaling protocol and the CS) obtain a prefix of the same global state or can identify an individual misbehaving party. This definition follows the game-based accountability property formalized for Hyperledger Fabric in [50] but using AUC takes it to the composable UC setting. As mentioned, we want to show this property for the composition of the scaling protocol and the consensus protocol.

(Assumption-based) liveness: Liveness states that transactions submitted by honest clients (of the scaling protocol and CS) will be part of the global state after at most δ time units assuming a network with bounded message delay (cf., e. g., [48, 50, 78, 80, 89]). To illustrate assumption-based properties, our analysis extends to the case that, at some point in the run, the underlying assumption of a network with bounded message delay might no longer hold true.

The ideal accountable consensus functionality $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$. To define $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$, we start by considering a (non-accountable) ideal consensus service functionality $\mathcal{F}_{\mathsf{cp}}$ that is essentially a simplified version of established ideal ledger functionalities, e. g., [10, 51], tailored towards the special case of consensus establishment. $\mathcal{F}_{\mathsf{cp}}$ enforces consistency and liveness as preventive security properties. We then apply AUC to obtain an accountable version $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ (cf., Figure 9 and 10 in Appendix D) that captures the above security properties. More specifically, we set $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{consistency}\}$ and $\mathsf{Sec}^{\mathsf{assumption}} = \{\mathtt{liveness}\}$. Note that $\mathtt{liveness} \notin \mathsf{Sec}^{\mathsf{acc}}$, and hence, $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ will not require judges to blame anybody (e.g., the network) if liveness fails.

We start by explaining $\mathcal{F}_{\mathsf{cp}}$ and then the AUC transformation to derive $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$. $\mathcal{F}_{\mathsf{cp}}$ itself consists of an unbounded number of clients who offer a read and write interface to higher-level protocols. These clients can write transactions to and read from a single globally ordered list/state in $\mathcal{F}_{\mathsf{cp}}$. Upon writing a new transaction, $\mathcal{F}_{\mathsf{cp}}$ models network traffic by allowing the simulator to determine when and in which order these transactions are appended to the global state. $\mathcal{F}_{\mathsf{cp}}$ guarantees that all incoming transactions by honest clients will be appended to the global state after at most δ time units. More formally, $\mathcal{F}_{\mathsf{cp}}$ models (absolute) preventive liveness by disallowing the attacker from advancing time as long as there are pending transactions that have not been added to the state since δ time units. Whenever $\mathcal{F}_{\mathsf{cp}}$ receives a read request from an honest client from a higher-level protocol, $\mathcal{F}_{\mathsf{cp}}$ allows $\mathcal{A}$ to determine the prefix of the global state that will then be returned to the client. For read requests received by corrupted/malicious clients, i. e., clients that are under full control of the adversary, $\mathcal{F}_{\mathsf{cp}}$ always allows $\mathcal{A}$ to freely determine the output of $\mathcal{F}_{\mathsf{cp}}$.

To derive $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ from $\mathcal{F}_{\mathsf{cp}}$, we implement the AUC transformation step $\mathcal{T}_2$ as follows. *(i)* $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ includes one incorruptible public judge (in a protocol session), i.e., $\mathsf{pids}_{\mathsf{judge}} = \{\mathtt{public}\}$, and considers only assumptions that affect all parties (in a session), i.e., $\mathsf{ids}_{\mathsf{assumption}} = \{\mathtt{public}\}$. *(ii)* As soon as liveness is marked broken (for $id = \mathtt{public}$), $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ no longer enforces that messages are added to the global state within δ time units. *(iii)* If (public) consistency is broken, $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ allows $\mathcal{A}$ to freely determine the output of $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ in turn for a fair

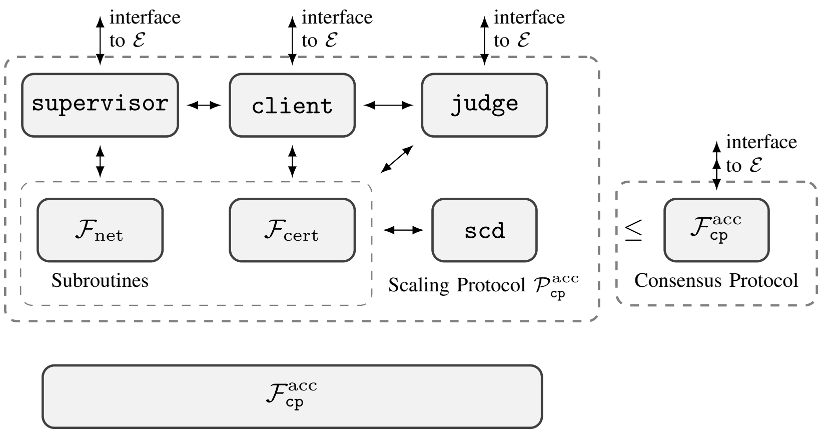

interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
supervisor
client
judge
interface to $\mathcal{E}$
$\mathcal{F}_{\text{net}}$
$\mathcal{F}_{\text{cert}}$
scd
$\leq$
$\mathcal{F}_{\text{cp}}^{\text{acc}}$
Subroutines
Scaling Protocol $\mathcal{P}_{\text{cp}}^{\text{acc}}$
Consensus Protocol
$\mathcal{F}_{\text{cp}}^{\text{acc}}$

Fig. 6: Illustration of Theorem 1. All machines are also connected to $\mathcal{A}$.

verdict matching the customization in $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$. The subroutine $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ forces $\mathcal{A}$ to provide a verdict which implies individual accountability, i. e., all parties in the verdict can rightfully be blamed for misbehavior. As long as there is no verdict reached yet in $\mathcal{F}_{\text{cp}}^{\text{acc}}$, $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$'s public judicial report returns a view on the global state which contains at least the longest prefix that was read by an honest client so far (everything else after this prefix can be chosen freely by $\mathcal{A}$). If there has been a verdict, the report is empty.

By this construction, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ indeed models the desired security properties *(individual public) accountability w.r.t consistency* and *assumption-based liveness.*

Protocol Model. We model the scaling protocol from Figure 5 via a hybrid protocol as depicted in Figure 6. Specifically, the `client` machine models the code run by the clients $\text{cl}_{i,j}$ and the internal machine `scd` models code of the SCD. The ideal subroutine $\mathcal{F}_{\text{cp}}^{\text{acc}}$ models an ideal accountable consensus protocol used by the scaling protocol, i. e., it abstracts the code of the CCD and the code of the consensus service CS, all of which are specified in a realization of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ (we discuss a possible realization at the end of this section). In a run of the protocol, there can be an unbounded number of instances of `client` and `scd`, each modeling one party in one protocol session. These parties additionally have access to an ideal signature functionality $\mathcal{F}_{\text{cert}}$ (which includes a PKI) and an ideal network functionality $\mathcal{F}_{\text{net}}$ with bounded message delay δ'. The adversary is allowed to break the assumption of a bounded network delay in $\mathcal{F}_{\text{net}}$ by sending a special message.

Observe that the CD consists of two components, SCD and CCD, modeled via separate machines, i. e., the server component `scd` and the client component in $\mathcal{F}_{\text{cp}}^{\text{acc}}$, but should be considered corrupted as soon as just one of those machines misbehaves. We capture this expected property by considering a party running `scd` to be corrupted also if the corresponding client in $\mathcal{F}_{\text{cp}}^{\text{acc}}$ is corrupted, i. e., we use the corruption state of `scd` to represent the corruption status of the combined CD. This idea allows for capturing individual accountability also in a composed protocol, where the same party takes part in multiple parts of a protocol and hence needs to be split into multiple machines. Without this mechanism the judge would be required, by individual accountability and fairness, to identify whether SCD or CCD has misbehaved. In addition to the above, since we identify a party (running `client` or `scd`) with its signature key, we also consider parties to be corrupted if their signature key is corrupted.

Two important aspects of the protocol model are the specifications of the public `judge` and the `supervisor`. The `judge` collects the evidence from clients and from the lower-level judge. That is, clients provide all sequences of messages/states that they received from SCDs (include SCD's signature over the state) to the judge as evidence. This judge also queries the public judge of the subroutine $\mathcal{F}_{\text{cp}}^{\text{acc}}$ to get the most recent verdicts and judicial report from $\mathcal{F}_{\text{cp}}^{\text{acc}}$. If the subroutine judge returns a non-empty verdict, then the `judge` outputs the same verdict since a misbehaving party was already found in the subroutine and hence consensus might no longer hold true. Otherwise, the `judge` verifies that evidence provided by clients is *(i)* correctly signed by a CD/SCD and *(ii)* the provided state is a prefix of the current judicial report, i. e., the correct consensus as determined by the public judge in the subroutine consensus protocol. If the first check fails, the client's evidence is not valid and is discarded. If the second check fails, the SCD violated consistency as it signed and forwarded a sequence of messages that differs from the established consensus. Hence, in this case the `judge` blames an individual SCD. In all other cases no misbehavior was detected and the verdict remains empty. For judicial reports, `judge` simply forwards the judicial report from the consensus protocol. The public judge follows the modeling as explained in Section 2.4 and, e. g., she reveals all gathered evidence to the network adversary.

The `supervisor` determines whether the assumptions needed for liveness still hold true by querying *(i)* $\mathcal{F}_{\text{net}}$ to check whether the bounded network delay is guaranteed and *(ii)* the supervisor of $\mathcal{F}^{\text{acc}}_{\text{cp}}$ to check whether liveness assumptions for the subroutine are still met (via `IsAssumptionBroken`?). If any of these checks fail, the supervisor returns that liveness assumptions no longer hold true. Hence, only in this case is the simulator in the ideal world allowed to actually break liveness.

UC Security Result. Our security result (cf., Figure 6) states that the scaling protocol $\mathcal{P}^{\text{acc}}_{\text{cp}}$ using an ideal accountable consensus service subroutine $\mathcal{F}^{\text{acc}}_{\text{cp}}$ is still an accountable consensus service, i. e., all security guarantees are retained and hence the scaled protocol is also a realization of $\mathcal{F}^{\text{acc}}_{\text{cp}}$:

Theorem 1. *Let $\mathcal{P}^{\text{acc}}_{\text{cp}}$ and $\mathcal{F}^{\text{acc}}_{\text{cp}}$ be as described above. Then,*

$$(\mathcal{P}^{\text{acc}}_{\text{cp}} \mid \mathcal{F}^{\text{acc}}_{\text{cp}}) \leq \mathcal{F}^{\text{acc}}_{\text{cp}}.$$

We provide the formal proof for Theorem 1 in Appendix D.

Discussion. Using the iUC composition theorem, the ideal subroutine $\mathcal{F}^{\text{acc}}_{\text{cp}}$ of $\mathcal{P}^{\text{acc}}_{\text{cp}}$ can be replaced with an arbitrary realization while retaining security results. The perhaps simplest realization consists of clients with access to a consensus service run by a single party, analogous to what is shown in Figure 5 (if CS were considered one party), where the consensus party signs its outputs to provide accountability. By Theorem 1, one can also realize the subroutine $\mathcal{F}^{\text{acc}}_{\text{cp}}$ via another copy of $\mathcal{P}^{\text{acc}}_{\text{cp}}$, i. e., one can iterate the above scaling approach to add additional scaling layers. Security of such a protocol with multiple scaling layers is then directly implied by Theorem 1 and the composition theorem. This nicely demonstrates one of the advantages of composition for accountability properties.

These kinds of composition results are enabled by several features offered by the AUC framework, most notably our novel concept of judicial reports. Indeed, if the public judge in the subroutine $\mathcal{F}^{\text{acc}}_{\text{cp}}$ had been unable to share his knowledge (i. e., a consistent view on the global state of the subroutine) via a judicial report with the higher-level `judge` in the scaling protocol, then the `judge` would be unable to decide, given two inconsistent views, whether CS or CD (CDs in the case of multiple layers) have provided inconsistent views. Without judicial reports, it would thus be impossible to prove individual accountability modularly.

As mentioned at the begin of this section, this case study illustrates the possibilities of AUC. Most notably, composability of accountability-based protocols enabled by judicial reports, the supervisor concept, handling of assumption-based and accountability properties concurrently, and blaming of internal parties (CD).

3.2 An Accountable PKI for the Web Based on CTLs

In our second case study, we analyze accountability of a Web PKI based on Certificate Transparency Logs (CTLs) [32, 72]. For the sake of presentation, we consider a slightly simplified version that does not include certificate revocation. We show that such a PKI with CTLs indeed achieves the expected property of *certificate transparency* [72], i. e., accountability w.r.t. certificate correctness. That is, if someone obtains a certified public key for Alice from the PKI, then either this key was indeed registered by Alice or Alice can identify and blame a misbehaving CA for issuing a wrong certificate based on the information provided by CTLs. In Section 3.3, we analyze and prove the security of a standard key exchange (KE) protocol composed with this accountable PKI protocol.

This case study along with the KE protocol uses several features of AUC, including some that were not yet illustrated in Section 3.1, such as *(i)* local accountability, *(ii)* individual and group-based accountability levels, and *(iii)* composition, including the case where higher-level judges use verdicts from lower level judges belonging to different parties. Altogether, using AUC, we are able to perform the first UC analysis of a CTL-based PKI and protocols based thereupon.

In what follows, we present our case study with full formal specifications available in Appendix E.

Protocol Description. As depicted in Figure 7, the roles in a CTL-based PKI protocol are *(i)* clients, *(ii)* CAs, and *(iii)* CTLs. Clients request CAs to issue a certificate for them and query CAs for certificates of other clients.

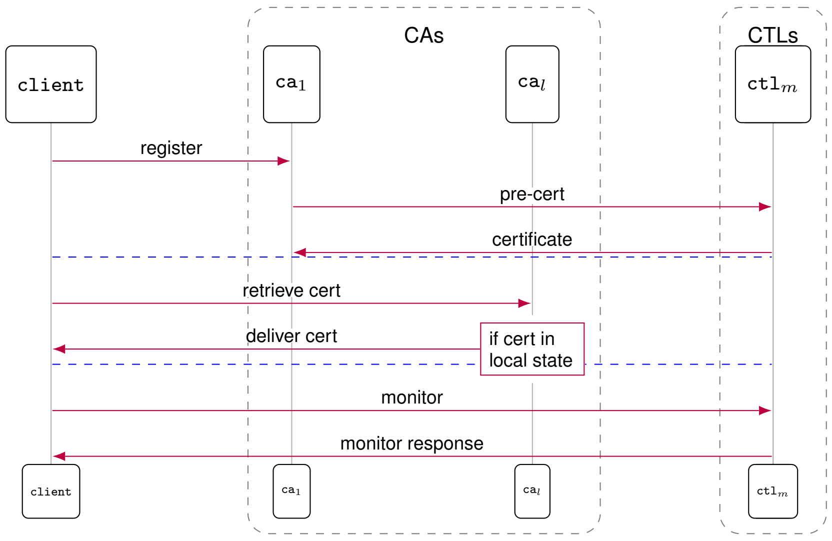


Fig. 7: CTL-based PKI protocol

Here we consider clients that certify at most one key.[5] More specifically, to certify a new key a client sends a registration request, containing its pid and the public key via an authenticated channel to a CA. When a CA receives a registration request, it checks that it did not issue a certificate for pid so far. If the request passes the check, the CA generates a pre-certificate containing the original request and a signature of the CA. The CA forwards the pre-certificate to potentially several CTLs. When a CTL receives a pre-certificate, it verifies the CA's signature. If the signature is valid, the CTL finalizes the certificate and also signs the certificate. CTLs store the certificates they signed/issued and allow clients to monitor these certificates. The underlying idea is that clients can detect identity theft by retrieving certificate lists from CTLs regularly and checking whether there are any certificates in their name that they did not request; we call these *maliciously created certificates* in what follows. Finally, a client pid can ask to obtain the certificate for another client pid' from a CA if such a certificate was issued by that CA.

Security Goal. Informally, a CTL-based PKI protocol such as the one presented here is supposed to achieve the security goal of *accountability w.r.t. certificate correctness* (typically called *certificate transparency*): honest parties detect maliciously generated certificates for their own identity after some bounded time delay. We denote this *local* accountability property in what follows by "`correctCert`". As long as `correctCert` holds true for a dedicated client, this means that the client actually requested the available certificate and there is no maliciously generated certificate available in the PKI.

The ideal PKI functionality $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. We formalize the property just sketched starting with a non-accountable ideal PKI functionality $\mathcal{F}_{\text{PKI}}$ that is analogous to Canetti et al.'s ideal functionality $\mathcal{G}_{\text{BB}}$ [27] and the ideal CA $\mathcal{F}_{\text{PKI}}$ from [17]. We then apply AUC to obtain an accountable version $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ that captures accountability w.r.t. certificate correctness.

Clients are the main roles of $\mathcal{F}_{\text{PKI}}$; CTLs and CAs are internal parties of potential realizations of $\mathcal{F}_{\text{PKI}}$. $\mathcal{F}_{\text{PKI}}$ allows (honest) clients to register one certificate for their own identity at some CA. The adversary $\mathcal{A}$ decides when and whether a registration is successful. If it succeeds, $\mathcal{F}_{\text{PKI}}$ issues the certificate (consisting of the party's pid, a string, meant to be the party's public key, and the name of the issuing CA) and adds the certificate to its state. When parties query $\mathcal{F}_{\text{PKI}}$ for a certificate of an honest pid' issued by a certain CA, $\mathcal{A}$ is free to choose when and whether $\mathcal{F}_{\text{PKI}}$ answers the request. If $\mathcal{A}$ instructs $\mathcal{F}_{\text{PKI}}$ to respond, $\mathcal{F}_{\text{PKI}}$ provides the unique certificate for pid' (as stored in $\mathcal{F}_{\text{PKI}}$'s state) or it outputs $\perp$ if there is no certificate recorded at that CA. For corrupted clients pid', $\mathcal{F}_{\text{PKI}}$ does not provide any guarantees but lets $\mathcal{A}$ freely determine the response.

[5]This is not an actual restriction. Higher-level protocols can certify multiple keys for a single identity pid' by setting, e. g., $pid = (pid', keyid)$.

We now apply AUC to $\mathcal{F}_{\mathrm{PKI}}$ to derive $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ which additionally captures local accountability w.r.t. certificate correctness. We include a local judge $((\mathtt{local}, pid, \mathtt{client}), sid, \mathtt{judge})$ for every client $(pid, sid, \mathtt{client})$ in $\mathcal{F}_{\mathrm{PKI}}$, i. e., $\mathsf{pids}_{\mathsf{judge}} \subset \{\mathtt{local}\} \times \{0,1\}^* \times \{\mathtt{client}\}$. We set $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{correctCert}\}$. This allows the adversary to indicate that certificate correctness is broken for some client pid as long as the adversary provides a verdict to the corresponding local judge. We require that these verdicts blame individual CAs, i. e., the affected client pid can identify those CAs that misissued a certificate in their name. This is enforced by instantiating $\mathcal{F}_{\mathrm{judgeParams}}^{\mathrm{acc\text{-}PKI}}$ to check that verdicts are of the form $\bigwedge_{i=1}^{n} \mathsf{dis}(CA_i)$, where CA_i are parties with the (internal) $\mathtt{ca}$ role. If $\mathtt{correctCert}$ is broken for a pid, then $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ treats pid in the same way as corrupted parties for the purpose of retrieving certificates, i. e., the adversary can return arbitrary certificates issued for pid.

Security model. We model the CTL-based PKI as a real protocol $\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}}$ which implements the previously mentioned roles and operations. We model dynamic sets of clients, CAs, and CTLs. Clients and CAs can be dynamically corrupted, whereas CTLs act as trust anchor and are hence incorruptible.[6] $\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}}$ uses an ideal signature functionality $\mathcal{F}_{\mathrm{sig}}$ for signatures. It further uses three ideal functionalities $\mathcal{F}_{\mathrm{init}}^{\mathsf{CA}}$, $\mathcal{F}_{\mathrm{psync\text{-}net}}$, $\mathcal{F}_{\mathrm{auth}}$ to capture setup assumptions: $\mathcal{F}_{\mathrm{init}}^{\mathsf{CA}}$ distributes public keys of CAs and CTLs, which are assumed to be known to all parties. $\mathcal{F}_{\mathrm{psync\text{-}net}}$ models a network with bounded message delay, i. e., messages are delivered within δ time units. This functionality is used by clients during certificate monitoring to guarantee that a response from the CTL is received in a timely fashion, thereby ensuring that clients can detect maliciously created certificates after some bounded time. $\mathcal{F}_{\mathrm{psync\text{-}net}}$ further provides a clock to all parties, capturing that parties are aware of the current time. Finally, $\mathcal{F}_{\mathrm{auth}}$ models an ideal authenticated channel which is used during certificate registration, modeling that a CA has some means to identify a client.

We model that a client pid regularly monitors CTL certificate lists, namely after at most δ time units (any other publicly known bound can be used as well). Hence, by the bounded message delay enforced by $\mathcal{F}_{\mathrm{psync\text{-}net}}$, we have that pid will detect a maliciously created certificate registered at some CTL after at most $3 \cdot \delta$. If the client pid detects such a certificate, then, from its point of view, the CA is at fault for signing a certificate that was not requested by her.[7] Hence, the local judge of pid blames such CAs via the verdict $\bigwedge_{i=1}^{n} \mathsf{dis}(CA_i)$, where CA_i are exactly those CAs that signed maliciously generated certificates for pid. Since a maliciously created certificate is detected by pid only after at most $3 \cdot \delta$ time units, any other party pid' that retrieves the certificate before that point in time cannot be sure whether a certificate is genuine or whether pid did not yet see and hence did not have the opportunity to complain about the certificate. We therefore model that clients, during certificate retrieval from some CA, accept certificates only with an age of at least $3 \cdot \delta$ time units. Such a certificate is either correct or pid' has already noticed and complained about the malicious certificate, as required by accountability w.r.t. certificate correctness.

Since we consider only local accountability, there is no public judge in $\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}}$. We also do not use judicial reports in this protocol. As we do not consider assumption-based security properties, we set $\mathsf{ids}_{\mathsf{assumption}} = \emptyset$. The supervisor in $\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}}$ is responsible only for forwarding the corruption status of the internal CAs. This ensures that a simulator in the ideal world can blame a CA in a verdict only if the CA is corrupted in the real world.

UC Security Result. We obtain the following result.

Theorem 2. *Let $\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}}$ be the real PKI protocol and $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ be the ideal accountable PKI functionality as described above. Then,*

$$\mathcal{P}_{\mathrm{PKI}}^{\mathrm{acc}} \leq \mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}.$$

We provide a formal proof of Theorem 2 in Appendix E.

3.3 A Key Exchange Based on an Accountable PKI

We now analyze a standard authenticated key exchange protocol, the so-called "ISO protocol", an authenticated version of the Diffie-Hellman key exchange with digital signatures based on the ISO/IEC 9798-3

[6]Trusting a CTL is indeed necessary as a malicious CTL can simply hide certificates during monitoring. This trust can be distributed among several CTLs by requiring $t \in \mathbb{N}$ CTLs to validate and sign new certificates. In this case, one can obtain a security result if at most $t-1$ CTLs are malicious. Our analysis carries over to this setting.

[7]Observe that only pid itself can be sure that the CA is at fault. A third party cannot determine whether the CA or pid has misbehaved (e. g., by requesting a certificate but then blaming an honest CA). This observation becomes important in the composed protocol analyzed in Section 3.3.

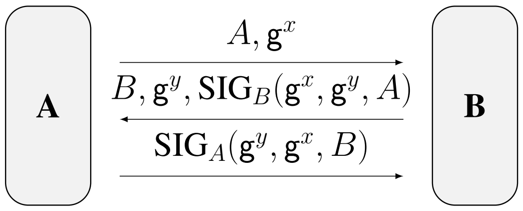

A
B
A, g^x
$B, g^y, \mathrm{SIG}_B(g^x, g^y, A)$
$\mathrm{SIG}_A(g^y, g^x, B)$

Fig. 8: The ISO protocol for mutually authenticated Diffie-Hellman key exchange between two parties A and B.

standard [56] (see Figure 8). UC security of this protocol has already been studied in various settings [17, 19, 20, 26, 27, 66] but always based on the assumption that the underlying PKI is perfect, i. e., the adversary cannot register certificates for honest parties. In contrast, we base our analysis on $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ which can then be realized by $\mathcal{P}^{\text{acc}}_{\text{PKI}}$ (Section 3.2) using the composition theorem. We thus provide the first analysis of the ISO protocol based on a PKI that may fail, but provides accountability when it does. This not only illustrates the features of AUC highlighted at the beginning of Section 3.2. This also shows that even protocols which traditionally consider only preventive security, such as key exchanges, can benefit from AUC. In the main body, we explain the main aspects of this case study; full details, including formal specifications and proofs, are provided in Appendix F.

Ideal accountable key exchange. A signature-based authenticated key exchange can only provide security as long as the underlying public signature key/certificate is trustworthy. We capture this intuition with AUC: We start with a standard ideal key exchange functionality $\mathcal{F}_{\text{KE}}$ [17, 20] and apply AUC to obtain an accountable version $\mathcal{F}^{\text{acc}}_{\text{KE}}$. In $\mathcal{F}^{\text{acc}}_{\text{KE}}$ we consider the local accountability property $\texttt{authenticity} \in \mathsf{Sec}^{\text{acc}}$ which determines if a party, say Alice, can still expect to authenticate her intended session partner, say Bob, or whether the certificate for Bob might be incorrect due to a fault in the PKI. If $\texttt{authenticity}$ is marked as broken for Alice by the adversary $\mathcal{A}$ (in exchange for a verdict), then $\mathcal{F}^{\text{acc}}_{\text{KE}}$ acts just as $\mathcal{F}_{\text{KE}}$ does in case one of the parties in the session is corrupted and hence no authentication can be guaranteed, i. e., it leaks the session key, if any, and allows $\mathcal{A}$ to determine the output for parties that have not yet finished the KE. We require verdicts to be of the form $\mathsf{dis}(p_{main}) \vee v$, where p_{main} is a main party, i. e., initiator or responder in this key exchange session, and v is a verdict containing only internal parties. This is a group based accountability level which captures that Alice in the real protocol, if Bob complains about a maliciously generated certificate, cannot decide whether Bob is lying or the PKI has actually misbehaved (cf. Footnote 7).

Protocol model. $\mathcal{P}^{\text{acc}}_{\text{iso}}$ is a straightforward model of the ISO protocol shown in Figure 8 derived from [17] consisting of the KE protocol part $\mathcal{P}^{\text{acc}}_{\text{pke}}$ and the ideal subroutine $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ which is used for public key distribution. As required by AUC, $\mathcal{P}^{\text{acc}}_{\text{iso}}$ contains local judges and a supervisor. As in Section 3.2, the supervisor only forwards the corruption status of internal parties including those that are part of $\mathcal{F}^{\text{acc}}_{\text{PKI}}$, namely $\mathcal{F}^{\text{acc}}_{\text{PKI}}$'s clients, CAs and CTLs. The main idea of the local judge of Alice is to request the verdict of Bobs local judge in $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. If this judge does not complain, then, by definition of $\mathcal{F}^{\text{acc}}_{\text{PKI}}$, any certificate of Bob that Alice retrieves must have been registered by Bob. If Bobs subroutine judge complains and returns a verdict v to Alice, then Alice's judge returns the overall verdict $\mathsf{dis}(p_{Bob}) \vee v$ (and vice versa for Bob's judge), capturing the aforementioned insecurity that Alice cannot decide whether Bob is lying or whether v is actually a fair verdict identifying a misbehaving PKI.

UC Security results. We can show the following:

Theorem 3. *Let $\mathcal{P}^{\text{acc}}_{\text{KE}}$, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$, and $\mathcal{F}^{\text{acc}}_{\text{KE}}$ be as described above and assume that the DDH assumption holds true. Then,*

$$(\mathcal{P}^{\text{acc}}_{\text{KE}} | \mathcal{F}^{\text{acc}}_{\text{PKI}}) \leq \mathcal{F}^{\text{acc}}_{\text{KE}}.$$

This immediately implies by UC composition that the key exchange remains secure when based on the real accountable CTL-based PKI $\mathcal{P}^{\text{acc}}_{\text{PKI}}$:

Corollary 4. *Let $\mathcal{P}^{\text{acc}}_{\text{KE}}$, $\mathcal{P}^{\text{acc}}_{\text{PKI}}$, and $\mathcal{F}^{\text{acc}}_{\text{KE}}$ be as described above. Then,*

$$(\mathcal{P}^{\text{acc}}_{\text{KE}} | \mathcal{P}^{\text{acc}}_{\text{PKI}}) \leq \mathcal{F}^{\text{acc}}_{\text{KE}}.$$

Proof. Follows from Theorem 2, Theorem 3, and the composition theorem of the underlying UC model. □

4. Related Work and Conclusion

As already discussed in Section 2.7, AUC focuses on property-based accountability, it generalizes and extends concepts from the literature and also introduces new concepts, such as judicial reports and supervisors to provide a general accountability framework.

Property-based Accountability in UC. To the best of our knowledge, the only other works that formalize and use the concept of property-based accountability in a UC model are those on MPC protocols, e.g., [13, 14, 24, 34, 35, 55, 74, 81]. As discussed in the introduction and in Appendix G, these works are specialized to the case of MPC and hence do not serve as general accountability frameworks. Most of these works consider composition of accountability properties with higher-level protocols to be out of scope. A notable exception is the very recent work of Baum et al. [14]. However, Baum et al. focus on the composability of verifiability in MPC protocols adhering to a specific structure. Baum et al. do not provide a general purpose accountability framework which can be used for arbitrary protocols.

Other simulation-based approaches. There are also a number of non-UC simulation-based formalizations of accountability properties, e.g., [5, 6, 8, 34, 37, 54, 86, 96]. Just as for the UC approaches mentioned above, these works analyze and are tailored towards MPC protocols and thus do not serve as general accountability frameworks. Furthermore, since they are not based on a UC model, they provide only weaker compositional properties, if any.

The covert adversaries model [6, 8] is perhaps the most prominent line of work in this category. The covert adversaries model formalizes accountability w.r.t. correctness (in the sense of identifiable abort) and w.r.t. privacy. AUC, even when restricted to the special case of MPC, and covert adversaries are incomparable due to different simulation paradigms. Both approaches can formalize accountability w.r.t. to correctness and privacy of MPC protocols (cf. Appendix G). On the one hand, AUC offers stronger composability that, unlike covert adversaries, also includes parallel composition. On the other hand, covert adversaries provide the additional concept of a deterrence factor ε to also model cases where a malicious party breaking security might remain undetected by a judge with (potentially non-negligible) probability ε. AUC models only the case that this probability is negligible. While it would be straightforward to add the same concept to AUC, we did not do so as it does not appear to offer any benefit within UC models. Indeed, it seems that all covert adversaries protocols that have been analyzed for a non-negligible ε use protocol rewinding within their simulators. This technique is not available to UC simulators since it prevents parallel composition, i.e., such protocols are not UC secure anyway. We leave further exploration of this aspect for future work.

Game-based accountability. There are many works that formalize accountability within a game-based setting, e.g., [42–46, 62, 63, 67, 79]. Some of these works are closely related to AUC in that they also consider highly general frameworks for accountability, e.g., [42, 45, 62, 67]. The main difference between AUC and these works is that AUC is the first general accountability framework for UC models, thereby providing particularly strong security statements while also offering the benefit of modular protocol analysis and composition. We, however, note that there are aspects in existing game-based accountability frameworks that AUC does not handle yet, such as *causality* [62]. It is an interesting challenge for future work to investigate whether and how these aspects can also be captured in a general accountability framework for UC.

Altogether, AUC lifts some of the work on game-based accountability frameworks to the UC setting, generalizes and unifies existing work on UC accountability, and also introduces several new concepts to make it a general purpose framework for accountability in UC.

5. Acknowledgments

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing".[8] This work was also supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through grant KU 1434/13-1.

[8]http://www.services-computing.de/

References

[1] B. Adida, "Helios: Web-based Open-Audit Voting," in *Proceedings of the 17th USENIX Security Symposium*, P. C. van Oorschot, Ed. USENIX Association, 2008, pp. 335–348.

[2] J. F. Almansa, I. Damgård, and J. B. Nielsen, "Simplified Threshold RSA with Adaptive and Proactive Security," in *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4004. Springer, 2006, pp. 593–611.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 2018, pp. 30:1–30:15.

[4] Apache Software Foundation, "Apache Kafka," https://kafka.apache.org/, 2017, (Accessed on 04/01/2019).

[5] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer Extensions," *J. Cryptol.*, vol. 30, no. 3, pp. 805–858, 2017.

[6] G. Asharov and C. Orlandi, "Calling Out Cheaters: Covert Security with Public Verifiability," in *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7658. Springer, 2012, pp. 681–698.

[7] N. Asokan, V. Shoup, and M. Waidner, "Asynchronous protocols for optimistic fair exchange," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1998, pp. 86–99.

[8] Y. Aumann and Y. Lindell, "Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries," in *Proceedings of the 4th Theory of Cryptography Conference,(TCC 2007)*, ser. Lecture Notes in Computer Science, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 137–156.

[9] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas, "Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC," in *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 12552. Springer, 2020, pp. 1–30.

[10] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a Transaction Ledger: A Composable Treatment," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10401. Springer, 2017, pp. 324–356.

[11] L. Baird and A. Luykx, "The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers," in *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*. IEEE, 2020, pp. 1–7.

[12] J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky, "Communication-Optimal Proactive Secret Sharing for Dynamic Groups," in *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 9092. Springer, 2015, pp. 23–41.

[13] C. Baum, I. Damgård, and C. Orlandi, "Publicly Auditable Secure Multi-Party Computation," in *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8642. Springer, 2014, pp. 175–196.

[14] C. Baum, E. Orsini, P. Scholl, and E. Soria-Vazquez, "Efficient Constant-Round MPC with Identifiable Abort and Public Verifiability," in *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 12171. Springer, 2020, pp. 562–592.

[15] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey, "Towards Better Availability and Accountability for IoT Updates by Means of a Blockchain," in *2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 50–58.

[16] V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget," *CoRR*, vol. abs/1710.09437, 2017.

[17] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch, "iUC: Flexible Universal Composability Made Simple," in *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 11923. Springer, 2019, pp. 191–221, the full version is available at http://eprint.iacr.org/2019/1073.

[18] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally Composable Security with Global Setup," in *Theory of Cryptography, Proceedings of TCC 2007*, ser. Lecture Notes in Computer Science, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 61–85.

[19] R. Canetti and J. Herzog, "Universally Composable Symbolic Analysis of Mutual Authentication and Key-Exchange Protocols," in *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006*, ser. Lecture Notes in Computer Science, S. Halevi and T. Rabin, Eds., vol. 3876. Springer, 2006, pp. 380–403.

[20] R. Canetti and H. Krawczyk, "Universally Composable Notions of Key Exchange and Secure Channels," in *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2332. Springer, 2002, pp. 337–351.

[21] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*. ACM Press, 2002, pp. 494–503.

[22] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[23] ——, "Universally Composable Security," *J. ACM*, vol. 67, no. 5, pp. 28:1–28:94, 2020.

[24] R. Canetti, A. Cohen, and Y. Lindell, "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9216. Springer, 2015, pp. 3–22.

[25] R. Canetti, K. Hogan, A. Malhotra, and M. Varia, "A Universally Composable Treatment of Network Time," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 360–375.

[26] R. Canetti and H. Krawczyk, "Security Analysis of IKE's Signature-Based Key-Exchange Protocol," in *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science, M. Yung, Ed., vol. 2442. Springer, 2002, pp. 143–161.

[27] R. Canetti, D. Shahaf, and M. Vald, "Universally Composable Authentication and Key-Exchange with Global PKI," in *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9615. Springer, 2016, pp. 265–296.
[28] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
[29] C. Chang and Y. Chang, "Efficient anonymous auction protocols with freewheeling bids," *Comput. Secur.*, vol. 22, no. 8, pp. 728–734, 2003.
[30] M. Ciampi, Y. Lu, and V. Zikas, "Collusion-Preserving Computation without a Mediator," Cryptology ePrint Archive, Tech. Rep. 2020/497, 2020.
[31] D. Ó. Coileáin and D. O'Mahony, "Accounting and Accountability in Content Distribution Architectures: A Survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 59:1–59:35, 2015.
[32] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, Internet Engineering Task Force, may 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt
[33] V. Cortier, D. Galindo, R. Küsters, J. Müller, and T. Truderung, "SoK: Verifiability Notions for E-Voting Protocols," in *IEEE 37th Symposium on Security and Privacy (S&P 2016).* IEEE Computer Society, 2016, pp. 779–798.
[34] R. K. Cunningham, B. Fuller, and S. Yakoubov, "Catching MPC Cheaters: Identification and Openability," in *Information Theoretic Security - 10th International Conference, ICITS 2017, Hong Kong, China, November 29 - December 2, 2017, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10681. Springer, 2017, pp. 110–134.
[35] E. Cuvelier and O. Pereira, "Verifiable Multi-party Computation with Perfectly Private Audit Trail," in *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, ser. Lecture Notes in Computer Science, vol. 9696. Springer, 2016, pp. 367–385.
[36] E. Cuvelier, O. Pereira, and T. Peters, "Election Verifiability or Ballot Privacy: Do We Need to Choose?" in *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8134. Springer, 2013, pp. 481–498.
[37] I. Damgård, C. Orlandi, and M. Simkin, "Black-Box Transformations from Passive to Covert Security with Public Verifiability," in *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 12171. Springer, 2020, pp. 647–676.
[38] G. D'Angelo, S. Ferretti, and M. Marzolla, "A Blockchain-based Flight Data Recorder for Cloud Accountability," in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems, CRYBLOCK@MobiSys 2018, Munich, Germany, June 15, 2018.* ACM, 2018, pp. 93–98.
[39] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl, "Globally Distributed Content Delivery," *IEEE Internet Comput.*, vol. 6, no. 5, pp. 50–58, 2002.
[40] Ethereum Foundation, "Ethereum enterprise," https://www.ethereum.org/enterprise/, 2019, (Accessed on 11/13/2019).
[41] C. Farkas, G. Ziegler, A. Meretei, and A. Lörincz, "Anonymity and accountability in self-organizing electronic communities," in *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society, WPES 2002, Washington, DC, USA, November 21, 2002.* ACM, 2002, pp. 81–90.
[42] J. Feigenbaum, "Privacy, Anonymity, and Accountability in Ad-Supported Services," in *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012.* IEEE Computer Society, 2012, pp. 9–10.
[43] J. Feigenbaum, J. A. Hendler, A. D. Jaggard, D. J. Weitzner, and R. N. Wright, "Accountability and deterrence in online life," in *Web Science 2011, WebSci '11, Koblenz, Germany - June 15 - 17, 2011.* ACM, 2011, pp. 7:1–7:7.
[44] J. Feigenbaum, A. D. Jaggard, and R. N. Wright, "Towards a formal model of accountability," in *2011 New Security Paradigms Workshop, NSPW '11, Marin County, CA, USA, September 12-15, 2011.* ACM, 2011, pp. 45–56.
[45] ——, "Open vs. closed systems for accountability," in *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014, Raleigh, NC, USA, April 08 - 09, 2014.* ACM, 2014, p. 4.
[46] ——, "Accountability in Computing: Concepts and Mechanisms," *Found. Trends Priv. Secur.*, vol. 2, no. 4, pp. 247–399, 2020.
[47] E. Funk, J. Riddell, F. Ankel, and D. Cabrera, "Blockchain technology: a data framework to improve validity, trust, and accountability of information exchange in health professions education," *Academic Medicine*, vol. 93, no. 12, pp. 1791–1794, 2018.
[48] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9057. Springer, 2015, pp. 281–310.
[49] S. Goldwasser and S. Park, "Public Accountability vs. Secret Laws: Can They Coexist?: A Cryptographic Proposal," in *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society, Dallas, TX, USA, October 30 - November 3, 2017.* ACM, 2017, pp. 99–110.
[50] M. Graf, R. Küsters, and D. Rausch, "Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric," in *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020.* Los Alamitos, CA, USA: IEEE, 2020, pp. 236–255.
[51] M. Graf, D. Rausch, V. Ronge, C. Egger, R. Küsters, and D. Schröder, "A Security Framework for Distributed Ledgers," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021.* New York City, USA: ACM, 2021, pp. 1043–1064.
[52] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 175–188.
[53] D. Hofheinz and V. Shoup, "GNUC: A New Universal Composability Framework," *J. Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.
[54] C. Hong, J. Katz, V. Kolesnikov, W. Lu, and X. Wang, "Covert Security with Public Verifiability: Faster, Leaner, and Simpler," in *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 11478. Springer, 2019, pp. 97–121.

[55] Y. Ishai, R. Ostrovsky, and V. Zikas, “Secure Multi-Party Computation with Identifiable Abort,” in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 8617. Springer, 2014, pp. 369–386.

[56] “ISO/IEC IS 9798-3, Entity authentication mechanisms — Part 3: Entity authentication using assymetric techniques,” 1993.

[57] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, “Towards a theory of accountability and audit,” in *ESORICS*. Springer, 2009.

[58] J. Kamto, L. Qian, J. Fuller, J. Attia, and Y. Qian, “Key Distribution and management for power aggregation and accountability in Advance Metering Infrastructure,” in *IEEE Third International Conference on Smart Grid Communications, SmartGridComm 2012, Tainan, Taiwan, November 5-8, 2012*. IEEE, 2012, pp. 360–365.

[59] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Capkun, “Misbehavior in Bitcoin: A Study of Double-Spending and Accountability,” *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 2:1–2:32, 2015.

[60] A. Kiayias, H. Zhou, and V. Zikas, “Fair and Robust Multi-party Computation Using a Global Transaction Ledger,” in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9666. Springer, 2016, pp. 705–734.

[61] T. H. Kim, L. Huang, A. Perrig, C. Jackson, and V. D. Gligor, “Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure,” in *22nd International World Wide Web Conference, WWW ’13, Rio de Janeiro, Brazil, May 13-17, 2013*. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 679–690.

[62] R. Künnemann, I. Esiyok, and M. Backes, “Automated Verification of Accountability in Security Protocols,” in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 397–413.

[63] R. Künnemann, D. Garg, and M. Backes, “Accountability in the Decentralised-Adversary Setting,” in *34th IEEE Computer Security Foundations Symposium, CSF 2021,*. IEEE, 2021.

[64] R. Küsters, “Simulation-Based Security with Inexhaustible Interactive Turing Machines,” in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006, pp. 309–320, see [71] for a full and revised version.

[65] R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt, “Ordinos: A Verifiable Tally-Hiding E-Voting System,” in *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 2020, pp. 216–235.

[66] R. Küsters and D. Rausch, “A Framework for Universally Composable Diffie-Hellman Key Exchange,” in *IEEE 38th Symposium on Security and Privacy (S&P 2017)*. IEEE Computer Society, 2017, pp. 881–900.

[67] R. Küsters, T. Truderung, and A. Vogt, “Accountability: Definition and Relationship to Verifiability,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, 2010, pp. 526–535, the full version is available at http://eprint.iacr.org/2010/236.

[68] R. Küsters, T. Truderung, and A. Vogt, “Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study,” in *32nd IEEE Symposium on Security and Privacy (S&P 2011)*. IEEE Computer Society, 2011, pp. 538–553.

[69] ——, “Clash Attacks on the Verifiability of E-Voting Systems,” in *33rd IEEE Symposium on Security and Privacy (S&P 2012)*. IEEE Computer Society, 2012, pp. 395–409.

[70] R. Küsters and M. Tuengerthal, “Composition Theorems Without Pre-Established Session Identifiers,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 41–50.

[71] R. Küsters, M. Tuengerthal, and D. Rausch, “The IITM model: a simple and expressive model for universal composability,” *Journal of Cryptology*, vol. 33, no. 4, pp. 1461–1584, 2020.

[72] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” RFC 6962, jun 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6962.txt

[73] H. Leibowitz, A. Herzberg, and E. Syta, “Provable Security for PKI Schemes,” Cryptology ePrint Archive, Tech. Rep. 2019/807, 2019.

[74] Y. Lindell and B. Pinkas, “Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer,” *J. Cryptol.*, vol. 25, no. 4, pp. 680–722, 2012.

[75] J. Liu, Y. Xiao, and J. Gao, “Achieving Accountability in Smart Grid,” *IEEE Syst. J.*, vol. 8, no. 2, pp. 493–508, 2014.

[76] S. Matsumoto and R. M. Reischuk, “Certificates-as-an-insurance: Incentivizing accountability in ssl/tls,” in *Proceedings of the NDSS Workshop on Security of Emerging Network Technologies (SENT’15)*, 2015.

[77] U. Maurer, “Constructive Cryptography - A New Paradigm for Security Definitions and Proofs,” in *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 6993. Springer, 2011, pp. 33–56.

[78] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The Honey Badger of BFT Protocols,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 31–42.

[79] K. Morio and R. Künnemann, “Verifying Accountability for Unbounded Sets of Participants,” in *34th IEEE Computer Security Foundations Symposium, CSF 2021,*. IEEE, 2021.

[80] R. Pass, L. Seeman, and A. Shelat, “Analysis of the Blockchain Protocol in Asynchronous Networks,” in *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 10211, 2017, pp. 643–673.

[81] A. Patra and D. Ravi, “Beyond Honest Majority: The Round Complexity of Fair and Robust Multi-party Computation,” in *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 11921. Springer, 2019, pp. 456–487.

[82] R3, “R3 Corda master documentation,” https://docs.corda.net/docs/corda-os/4.4.html, 2020, (Accessed on 04/24/2020).

[83] H. V. Ramasamy, A. Agbaria, and W. H. Sanders, “A Parsimonious Approach for Obtaining Resource-Efficient and Trustworthy Execution,” *IEEE Trans. Dependable Secur. Comput.*, vol. 4, no. 1, pp. 1–17, 2007.

[84] K. Ramchen, C. Culnane, O. Pereira, and V. Teague, “Universally Verifiable MPC and IRV Ballot Counting,” in *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 11598. Springer, 2019, pp. 301–319.

[85] M. Rivinius, P. Reisert, D. Rausch, and R. Küsters, “Publicly accountable robust multi-party computation,” in *S&P ’22*. IEEE, 2022.

[86] B. Schoenmakers and M. Veeningen, “Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems,” in *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 9092. Springer, 2015, pp. 3–22.
[87] P. Scholl, M. Simkin, and L. Siniscalchi, “Multiparty Computation with Covert Security and Public Verifiability,” Cryptology ePrint Archive, Tech. Rep. 2021/366, 2021.
[88] A. Shamis, P. Pietzuch, B. Canakci, M. Castro, C. Fournet, E. Ashton, A. Chamayou, S. Clebsch, A. Delignat-Lavaud, M. Kerner *et al.*, “IA-CCF: Individual accountability for permissioned ledgers,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 467–491.
[89] C. Stathakopoulou, T. David, and M. Vukolic, “Mir-BFT: High-Throughput BFT for Blockchains,” *CoRR*, vol. abs/1906.05552, 2019.
[90] Y. S. Tan, R. K. L. Ko, and G. Holmes, “Security and Data Accountability in Distributed Systems: A Provenance Survey,” in *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*. IEEE, 2013, pp. 1571–1578.
[91] The Guardian, “Steve jobs suggests: get rid of the drm on online music,” https://www.theguardian.com/technology/blog/2007/feb/06/stevejobssugg, 2007, (Accessed on 07/09/2021).
[92] VASCO Data Security International, Inc., “news: Vasco announces bankruptcy filing by diginotar b.v.” https://web.archive.org/web/20110923180445/http://www.vasco.com/company/press_room/news_archive/2011/news_vasco_announces_bankruptcy_filing_by_diginotar_bv.aspx, 09 2011, (Accessed on 06/13/2022).
[93] WIRED, “Shocker: Apple’s drm-free music not so easily stolen,” https://www.wired.com/2007/06/apples-drmfree-/, 2007, (Accessed on 07/09/2021).
[94] Z. Xiao, N. Kathiresshan, and Y. Xiao, “A survey of accountability in computer networks and distributed systems,” *Secur. Commun. Networks*, vol. 9, no. 4, pp. 290–315, 2016.
[95] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, “Separating agreement from execution for byzantine fault tolerant services,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. ACM, 2003, pp. 253–267.
[96] B. Zeng, C. Tartary, P. Xu, J. Jing, and X. Tang, “A Practical Framework for t-Out-of-n Oblivious Transfer With Security Against Covert Adversaries,” *IEEE Trans. Inf. Forensics Secur.*, vol. 7, no. 2, pp. 465–479, 2012.
[97] Z. Zhao, M. Naseri, and Y. Zheng, “Secure quantum sealed-bid auction with post-confirmation,” *Optics Communications*, vol. 283, no. 16, pp. 3194–3197, 2010.
[98] J. Zhou and D. Gollmann, “A fair non-repudiation protocol,” in *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1996, pp. 55–61.
[99] ——, “An Efficient Non-repudiation Protocol,” in *10th Computer Security Foundations Workshop (CSFW ’97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 1997, pp. 126–132.
[100] ——, “Evidence and non-repudiation,” *Journal of Network and Computer Applications*, vol. 20, no. 3, pp. 267–281, 1997.

Appendix

A. A Brief Introduction to the iUC Framework

This section provides a brief introduction to the iUC framework, which underlies all results in this paper. The iUC framework [17] is a highly expressive and user-friendly model for universal composability. It allows for the modular analysis of different types of protocols in various security settings.

The iUC framework uses interactive Turing machines as its underlying computational model. Such interactive Turing machines can be connected to each other to be able to exchange messages. A set of machines $\mathcal{Q} = \{M_1, \ldots, M_k\}$ is called a *system*. In a run of $\mathcal{Q}$, there can be one or more instances (copies) of each machine in $\mathcal{Q}$. One instance can send messages to another instance. At any point in a run, only a single instance is active, namely, the one to receive the last message; all other instances wait for input. The active instance becomes inactive once it has sent a message; then the instance that receives the message becomes active instead and can perform arbitrary computations. The first machine to run is the so-called *master*. The master is also triggered if the last active machine did not output a message. In iUC, the environment (see next) takes the role of the master. In the iUC framework a special user-specified **CheckID** algorithm is used to determine which instance of a protocol machine receives a message and whether a new instance is to be created (see below).

To define the universal composability security experiment (cf. [17]), one distinguishes between three types of systems: protocols, environments, and adversaries. As is standard in universal composability models, all of these types of systems have to meet a polynomial runtime notion. Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be “as good as” $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$). The environment can also subsume the role of the network attacker $\mathcal{A}$, which yields an equivalent definition in the iUC framework. We usually show this equivalent but simpler statement in our proofs, i.e., that there exists a simulator $\mathcal{S}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all environments.

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$; the framework offers a convenient template for the specification of such systems. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol $\mathcal{P}_{\text{sig}}$ for digital signatures might contain a `signer` role for signing messages and a `verifier` role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment subsuming a network attacker) via network interfaces. An instance of a machine M_i manages one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) pid running in a session with session ID (SID) sid and executing some code defined by the role $role$ where this role has to be (one of) the role(s) of M_i according to the specification of M_i. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. More specifically, the I/O interfaces of both machines need to be connected to each other (because one machine specifies the other as a subroutine) to enable communication between entities of those machines.

Roles of a protocol can be either public or private. The I/O interfaces of private roles are only accessible by other (entities belonging to) roles of the same protocol, whereas I/O interfaces of public roles can also be accessed by other (potentially unknown) protocols/the environment. Hence, a private role models some internal subroutine that is protected from access outside of the protocol, whereas a public role models some publicly accessible operation that can be used by other protocols. One uses the syntax "$(\texttt{pubrole}_1, \ldots, \texttt{pubrole}_\texttt{n} \mid \texttt{privrole}_1, \ldots, \texttt{privrole}_\texttt{n})$" to uniquely determine public and private roles of a protocol. Two protocols $\mathcal{P}$ and $\mathcal{Q}$ can be combined to form a new more complex protocol as long as their I/O interfaces connect only via their public roles. In the context of the new combined protocol, previously private roles remain private while previously public roles may either remain public or be considered private, as determined by the protocol designer. The set of all possible combinations of $\mathcal{P}$ and $\mathcal{Q}$, which differ only in the set of public roles, is denoted by $\mathsf{Comb}(\mathcal{Q}, \mathcal{P})$.

An entity in a protocol might become corrupted by the adversary, in which case it acts as a pure message forwarder between the adversary and any connected higher-level protocols as well as subroutines. In addition, an entity might also consider itself (implicitly) corrupted while still following its own protocol because, e.g., a subroutine has been corrupted. Corruption of entities in the iUC framework is highly customizable; one can, for example, prevent corruption of certain entities during a protected setup phase.

The iUC framework supports the modular analysis of protocols via a so-called composition theorem:

Corollary 5 (Concurrent composition in iUC; informal)**.** *Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ can be connected. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ agree on their public roles. Then $\mathcal{R} \leq \mathcal{I}$.*

By this theorem, one can first analyze and prove the security of a subroutine $\mathcal{P}$ independently of how it is used later on in the context of a more complex protocol. Once we have shown that $\mathcal{P} \leq \mathcal{F}$ (for some other, typically ideal protocol $\mathcal{F}$), we can then analyze the security of a higher-level protocol $\mathcal{Q}$ based on $\mathcal{F}$. Note that this is simpler than analyzing $\mathcal{Q}$ based on $\mathcal{P}$ directly as ideal protocols provide absolute security guarantees while typically also being less complex, reducing the potential for errors in proofs. Once we have shown that the combined protocol, say, $(\mathcal{Q} \mid \mathcal{F})$ realizes some other protocol, say, $\mathcal{F}'$, the composition theorem and transitivity of the $\leq$ relation then directly implies that this also holds true if we run $\mathcal{Q}$ with an implementation $\mathcal{P}$ of $\mathcal{F}$. That is, $(\mathcal{Q} \mid \mathcal{P})$ is also a secure realization of $\mathcal{F}'$. Please note that the composition theorem does not impose any restrictions on how the protocols $\mathcal{P}$, $\mathcal{F}$, and $\mathcal{Q}$ look like internally. For example, they might have disjoint sessions, but they could also freely share some state between sessions, or they might be a mixture of both. They can also freely share some of their subroutines with the environment, modeling so-called globally available state. This is unlike most other models for universal composability, such as the UC model, which impose several conditions on the structure of protocols for their composition theorem.

B. Notation in Pseudo Code

ITMs in our paper are specified in pseudo code. Most of our pseudo code notation follows the notation introduced by Camenisch et al. [17]. To ease readability of our figures, we provide a brief overview over the used notation here.

The description in the main part of the ITMs consists of blocks of the form **recv** ⟨*msg*⟩ **from** ⟨*sender*⟩ **to** ⟨*receiver*⟩ **s.t.** ⟨condition⟩:⟨code⟩ where ⟨*msg*⟩ is an input pattern, ⟨*sender*⟩ is the receiving interface (I/O or NET), ⟨*receiver*⟩ is the dedicated receiver of the message and ⟨condition⟩ is a condition on the input. ⟨code⟩ is the (pseudo) code of this block. The block is executed if an incoming message matches the pattern and the condition is satisfied. More specifically, ⟨*msg*⟩ defines the format of the message m that invokes this code block. Messages contain local variables, state variables, strings, and maybe special characters. To compare a message m to a message pattern *msg*, the values of all global and local variables (if defined) are inserted into the pattern. The resulting pattern p is then compared to m, where uninitialized local variables match with arbitrary parts of the message. If the message matches the pattern p and meets ⟨condition⟩ of that block, then uninitialized local variables are initialized with the part of the message that they matched to and ⟨code⟩ is executed in the context of ⟨*receiver*⟩; no other blocks are executed in this case. If m does not match p or ⟨condition⟩ is not met, then m is compared with the next block. Usually a **recv from** block ends with a **send to** clause of form **send** $\langle\overline{msg}\rangle$ **to** $\langle\overline{sender}\rangle$ where $\overline{msg}$ is a message that is send via output interface $\overline{sender}$.

If an ITM invokes another ITM, e. g., as a subroutine, ITMs may expect an immediate response. In this case, in a **recv from** block, a **send to** statement is directly followed by a **wait for** statement. We write **wait for** $\langle\overline{msg}\rangle$ **from** $\langle\overline{sender}\rangle$ **s.t.** ⟨*condition*⟩ to denote that the ITM stays in its current state and discards all incoming messages until it receives a message m matching the pattern $\overline{msg}$ and fulfilling the **wait for** condition. Then the ITM continues the run where it left of, including all values of local variables.

To clarify the presentation and distinguish different types of variables, constants, strings, etc. we follow the naming conventions of Camenisch et al. [17]:

1. (Internal) state variables are denoted by sans-serif fonts.
2. Local (i.e., ephemeral) variables are denoted in *italic font*.
3. Keywords are written in **bold font** (e. g., for operations such as **sending** or **receiving**).
4. Commands, procedure, function names, strings and constants are written in `teletype`.

To increase readability, we use the following notation:

- For a set of tuples K, K.`add`(_) adds the tuple to K.
- For a string S, S.`add`(_) concatenates the given string to S.
- For a verdicts v_1 and v_2, we define v_1.`add`$(v_2) := v_1 \wedge v_2$.
- K.`remove`(_) removes always the first appearance of the given element/string from the list/tuple/set/string K.

We use the following additional nomenclature from [17]:

- $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ denotes the currently active entity and $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ denotes the entity which called the currently active ITM.
- The macro $\mathbf{corr}(pid, sid, role)$ is simply a shortcut to invoke the ITM of $(pid, sid, role)$ and query it for its corruption status.
- The macro $\mathbf{init}(pid, sid, role)$ triggers the initialization of $(pid, sid, role)$ and returns the activation to the calling ITM.

C. Reusing Existing Security Results with AUC

In this section, we briefly discuss how existing security results can be reused when transforming an ideal functionality with AUC. Therefore, we consider an ideal functionality $\mathcal{F}$ which ensures some security properties $\mathcal{S}ec$ and its accountable transformation $\mathcal{F}^{\text{acc}}$ (cf. Definition 1) which may replace some of $\mathcal{F}$'s security properties with the corresponding assumption-based and/or accountability properties, i. e., $\mathcal{S}ec^{\text{abs}} \cup \mathcal{S}ec^{\text{assumption}} \cup \mathcal{S}ec^{\text{acc}} = \mathcal{S}ec$. According to Definition 1, $\mathcal{F}$ and $\mathcal{F}^{\text{acc}}$ behave identical as long as no security property is marked as broken. Thus, an existing security proof remains valid until this case occurs. Thus, one often can reuse the existing security proof as is (up to the point where some property breaks). As AUC is a non-back-box transformation, one needs to enhance proofs in order to capture the behavior when transforming $\mathcal{F}$ to $\mathcal{F}^{\text{acc}}$.

We further note that accountability/assumption-based and preventive security properties can be heavily intertwined. Therefore one may also need to adjust the existing simulator to correctly handle broken properties of $\mathcal{F}^{\text{acc}}$.

We essentially illustrate this approach in our case study in Section 3.3.

D. Scaling Accountable Consensus (Full Details)

In this section, we provide full details regarding the scalable accountable consensus case study presented in Section 3.1. In particular, we provide *(i)* a full specification of $\mathcal{F}_{\text{cp}}^{\text{acc}}$, *(ii)* a full specification of the scaling protocol, and *(iii)* provide a formal proof for Theorem 1.

D.1 The Accountable Consensus Functionality $\mathcal{F}_{\text{cp}}^{\text{acc}}$: In this section, we present a full specification of the ideal accountable consensus service $\mathcal{F}_{\text{cp}}^{\text{acc}}$ in Figure 9 and 10 including its subroutine $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ in Figure 11. For technical details and notation specific to the iUC framework, see the brief summary of the iUC framework in Section A.

$\mathcal{F}_{\text{cp}}^{\text{acc}}$ models one instance of a consensus protocol per sid. $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s duty is to allow clients to write to a stream of messages (also called the state of $\mathcal{F}_{\text{cp}}^{\text{acc}}$) and read from this stream. For write operations, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ expects messages of the form $(\texttt{Submit}, msg)$ where msg denotes some arbitrary bit string. Submitted transactions/messages do not enter the state of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ immediately. Firstly, they get an unambiguous temporary ID and are then stored in a buffer buffer for later processing - this is necessary to capture network artifacts. Besides the submitted message, the buffer also contains the round, resp. point in time (provided by $\mathcal{F}_{\text{cp}}^{\text{acc}}$ internal clock, see below), when the message was added to the buffer. This will later be used to enforce liveness. The submitted message as well as its temporary ID are leaked to $\mathcal{A}$. Note that the `Submit` interface is accessible via I/O and NET. The adversary $\mathcal{A}$ is allowed to read the buffer via the `getBuffer` command. Further, $\mathcal{A}$ is allowed to permutate the content of the buffer, i.e., $\mathcal{A}$ may interchange the order/IDs in the buffer. To add the first element from buffer to $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s state, $\mathcal{A}$ can use the `Update` command. When an entry from buffer is moved to state, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ assigns an incremental counter as ID to the message and removes the message from the buffer. That is, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ establishes a total order over the transactions in state.

When a party $(pid, sid, role)$ wants to read $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s current state, it sends `Read` via I/O to $\mathcal{F}_{\text{cp}}^{\text{acc}}$. Analogously to submit requests, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ buffers the request in buffer_{read} including an unambiguous ID id and the current time. $\mathcal{A}$ is then expected to trigger the delivery of the response to the request via $(\texttt{Deliver}, id)$. If the accountability property `consistency` is already broken, $\mathcal{A}$ can freely determine $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s responses to the read request. To do so, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ queries $\mathcal{A}$ for the response and forwards the output to the requestor. If `consistency` still holds true, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ requests $\mathcal{A}$ for the prefix of the state it should provide to the requestor. More specifically, $\mathcal{A}$ provides a pointer N to the prefix of the state and a potential receiver $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_{)}$ to $\mathcal{F}_{\text{cp}}^{\text{acc}}$. If there exists an entry in buffer_{read} matching id, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ ignores the receiver provided by $\mathcal{A}$ and sends the prefix of the state to the requesting party stored in buffer_{msg} (for id) and deletes the read request from buffer_{read}. If there is no corresponding message to id in buffer_{read}, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ pushes the prefix of the state to $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r)$.

As already mentioned, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ also includes an (internal) clock which allows measuring time to capture liveness properties. Parties and $\mathcal{A}$ can query the current time of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ via the `GetCurRound` command. Further, $\mathcal{A}$ is allowed to increase the round, resp. current time, in $\mathcal{F}_{\text{cp}}^{\text{acc}}$ by one via `UpdateRound`. In case that `liveness` still holds true, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ checks that there does not exist an entry (id, msg, r) in buffer such that it holds true $\mathsf{round} - \mathsf{r} > \delta$ where round is the current round in $\mathcal{F}_{\text{cp}}^{\text{acc}}$ and δ is the liveness parameter of $\mathcal{F}_{\text{cp}}^{\text{acc}}$. If `liveness` is already broken, there are no restrictions on time updates.

In $\mathcal{F}_{\text{cp}}^{\text{acc}}$, we use the standard corruption behavior for dynamic corruption without secure erasures of iUC. In particular, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ only acts as message forwarder for corrupted parties. Further, we model a dynamic set of participants in $\mathcal{F}_{\text{cp}}^{\text{acc}}$.

$\mathcal{F}_{\text{cp}}^{\text{acc}}$ also relies on the concepts of AUC in order to capture and handle accountability properties. We expect $\mathcal{F}_{\text{cp}}^{\text{acc}}$ to be used with the parameters $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{consistency}\}$ and $\mathsf{Sec}^{\mathsf{assumption}} = \{\texttt{liveness}\}$. For the security properties, $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ enforces that $\mathcal{A}$ has to trade in a (public) verdict for $pid_j = \texttt{public}$, i.e., we expect the parameter $\mathsf{pids}_{\mathsf{judge}}$ to be $\{\texttt{public}\}$, which ensures individual accountability if he breaks `consistency`. To capture that liveness may break for all participating parties, we expect $\mathsf{ids}_{\mathsf{assumption}}$ to be $\{\texttt{public}\}$. $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ does not provide additional leakage to $\mathcal{A}$ if he breaks `liveness`. As judicial report $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ outputs a prefix of $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s state including an extension provided by $\mathcal{A}$. In particular, the prefix needs to contain at least

Description of the ideal and accountable consensus service $\mathcal{F}_{\text{cp}}^{\text{acc}} = (\texttt{client}, \texttt{judge}, \texttt{supervisor})$:

```
Participating roles: {client, judge, supervisor}
Corruption model: dynamic corruption without secure erasures
Protocol parameters:
    – $\delta \in \mathbb{N}$                                   {The expected liveness guarantee for in time units
    – $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$                    {Accountability properties
    – $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$             {Assumption-based security properties
    – $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$      {set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)
    – $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$          {set of entities/IDs where properties are ensured via assumptions
```

Description of $M_{\text{cp}}^{\text{acc}}$:

```
Implemented role(s): {client}
Subroutines: $\mathcal{F}_{\text{judgeParams}}^{\text{acc-cp}}$ : judgeParams
Internal state:
    – $\mathsf{state} \subset \mathbb{N} \times \{0,1\}^*$, $\mathsf{state} = \emptyset$        {The set of totally ordered transactions/messages
    – $\mathsf{buffer} \subset \mathbb{N} \times \{0,1\}^* \times \mathbb{N}$, $\mathsf{buffer} = \varepsilon$     {The buffer ob submitted transactions including submission round
    – $\mathsf{buffer}_{read} \subset \mathbb{N}^2 \times (\{0,1\}^*)^3$, $\mathsf{buffer}_{read} = \emptyset$   {Buffer for handling Read requests, entries of form $(id, round, (pid, sid, role))$
    – $\mathsf{counter}, \mathsf{counter}_B, \mathsf{counter}_R \in \mathbb{N}$, $\mathsf{counter} = 0$, $\mathsf{counter}_B = 0$, $\mathsf{counter}_R = 0$   {The counter for odered transactions, to order tx in buffer, and read requests in $\mathsf{buffer}_{read}$
    – $\mathsf{round} \in \mathbb{N}$, $\mathsf{round} = 0$                      {The time in $\mathcal{F}_{\text{cp}}^{\text{acc}}$
    – $\mathsf{maxHonestOutput} \in \mathbb{N}$, $\mathsf{maxHonestOutput} = 0$   {The longest prefix of the state queried by an honest party
    – $\mathsf{corruptedIntParties} \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\mathtt{judge}, \mathtt{supervisor}\})$, initially $\emptyset$   {The set of corrupted internal parties $(pid, sid, role)$
    – $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\mathtt{true}, \mathtt{false}\}$ {Stores broken security assumptions per id, initially false ∀entries
    – $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\mathtt{true}, \mathtt{false}\}$   {Stores broken security properties per judge/id, initially false ∀entries
    – $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$          {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$. For messages to $(pid, sid, \mathtt{judge})$ accept only if $pid = \mathtt{public}$.
Corruption behavior:
    – AllowCorruption($pid, sid, role$):
        Do not allow corruption of $(pid, sid, \mathtt{supervisor})$.
        if $role = \mathtt{judge}$:
            send $(\mathsf{Corrupt}, (pid, sid, \mathtt{judge}), \mathsf{internalState})$      {$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted
              to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \mathtt{judgeParams})$
            wait for $b$; return $b$
    – DetermineCorrStatus($pid, sid, role$):
        if $role = \mathtt{judge}$:                    {$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status
            send $(\mathsf{CorruptionStatus?}, (pid, sid, \mathtt{judge}), \mathsf{internalState})$ to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \mathtt{judgeParams})$
            wait for $b$; return $b$
    – AllowAdvMessage($pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m$)
        Do not allow sending messages to $\mathcal{F}_{\text{judgeParams}}$.   {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}_{\text{judgeParams}}$ in the name of corrupted parties.
Main:
        recv $(\mathtt{Submit}, msg)$:                                     {Transaction submission
            $\mathsf{counter}_B \leftarrow \mathsf{counter}_B + 1$; $\mathsf{buffer.add}([\mathsf{counter}_B, msg, \mathsf{round}])$
            send $(\mathtt{Submit}, \mathsf{counter}_B, msg)$ to NET                 {Leak submitted data to $\mathcal{A}$
        recv Read from I/O:                                                {Read request
            $\mathsf{counter}_R \leftarrow \mathsf{counter}_R + 1$; $\mathsf{buffer}_{read}.\mathsf{add}(\mathsf{counter}_R, \mathsf{round}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}))$  {Record read request including current time
            send $(\mathtt{Read}, \mathsf{counter}_R, \mathsf{round}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}))$ to NET   {Leak information to $\mathcal{A}$
        recv $(\mathtt{Deliver}, id)$ from NET:     {$\mathcal{A}$ triggers delivery of response to read request or injects a response without a read request
            if $\mathsf{brokenProps}[\mathtt{consistency}, \mathtt{public}] = \mathtt{true}$:   {Check whether $\mathcal{F}_{\text{cp}}^{\text{acc}}$ still provides consistency
                send responsively Read to NET      {If consistency is "broken" $\mathcal{A}$ is allowed to determine the output of $Facs$
                wait for $(\mathtt{Read}, state, (pid, sid, role))$
                if $\exists (id, round, (pid, sid, role)) \in \mathsf{buffer}_{msg}$:    {Check that ID exists
                    $\mathsf{maxHonestOutput} \leftarrow |\mathsf{state}|$               {Record longest output of the state so far
                    $\mathsf{buffer}_{read}.\mathtt{remove}([id, round, (pid, sid, role)])$; send $(\mathtt{Read}, state)$ to $(pid, sid, role)$
            else:                                                      {If consistency holds, output the current state
                send responsively Read to NET   $(\star)$           {Allow $\mathcal{A}$ to determine the prefix of state as response
                wait for $(\mathtt{Read}, N, (pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r))$     {$\mathcal{A}$ decides on the prefix of the state. If there is no read request for id, $\mathcal{A}$ also determines the recipient.
                if $|\mathsf{state}| < N$:
                    Go to $(\star)$
                if $\exists (id, round, (pid, sid, role)) \in \mathsf{buffer}_{msg}$:         {Check that ID exists
                    $\mathsf{buffer}_{read}.\mathtt{remove}([id, round, (pid, sid, role)])$; Let $state$ be the prefix of state up to and including entry $N$.
                    send $(\mathtt{Read}, state)$ to $(pid, sid, role)$            {Send state to requestor
                else:
                    send $(\mathtt{Read}, state)$ to $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r)$      {Send state the entity determined by $\mathcal{A}$
        recv Read from NET:                         {$\mathcal{A}$ is allowed to access the full state of $\mathcal{F}_{\text{cp}}^{\text{acc}}$.
            reply $(\mathtt{Read}, \mathsf{state})$
        recv getBuffer from NET:                    {$\mathcal{A}$ is allowed to query the current buffer
            reply $(\mathtt{getBuffer}, \mathsf{buffer})$
        recv $(\mathtt{Permute}, \pi)$ from NET:    {$\mathcal{A}$ is allowed to change the order in the buffer
            require: $\pi$ is a permutation of the IDs in buffer.
            for all $(ctr, msg) \in \mathsf{buffer}$ do: $\mathsf{buffer}.\mathtt{remove}([ctr, msg])$; $\mathsf{buffer}.\mathtt{add}([\pi(ctr), msg])$
            reply ack
```

$^{a}\mathsf{Roles}_{\mathcal{F}} = \{\mathtt{client}\}$ here.

Fig. 9: The ideal and accountable consensus service $\mathcal{F}_{\text{cp}}^{\text{acc}}$ including accountability (Part 1).

Description of $M^{\text{acc}}_{\text{cp}}$ (continued):

```
Main:
    recv Update from NET:                                   {A may trigger when buffer is included in state
        Let $(i, msg) \in$ buffer the first entry in buffer; counter $\leftarrow$ counter $+ 1$; state.add([counter, $msg$]); buffer.remove([i, $msg$])
                                                                                    {Remove entry from buffer
        reply ack
    recv UpdateRound from NET:                                                      {$\mathcal{A}$ triggers round update
        if $(\exists(\_, \_, r, ) \in$ buffer,
          s.t. round $-$ r $> \delta \vee \exists(\_, $r$, \_) \in$ buffer$_{read}$,
          s.t. round $-$ r $> \delta) \wedge$ brokenProps[liveness, public] $=$ false:
            reply (UpdateRound, false, $\epsilon$)      {Reject round update if there are old messages in queue but liveness still holds
        else:
            round $\leftarrow$ round $+ 1$
            reply (UpdateRound, true, $\epsilon$)
    recv GetCurRound:                                                     {Allow access to the "global clock"
        reply (GetCurRound, round)
    Include static code from the AUC transformation $\mathcal{T}_1(\cdot)$ here, i.e., include additional code from Figure 2 and 3 here.
```

Fig. 10: The ideal and accountable consensus service $\mathcal{F}^{\text{acc}}_{\text{cp}}$ including accountability (Part 2).

Description of $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}} = (\text{judgeParams})$:

```
Participating roles: {judgeParams}
Corruption model:    incorruptible
```

Description of $M^{\text{acc-aCS}}_{\text{judgeParams}}$:

```
Implemented role(s): {judgeParams}
Internal state:
    – $ptr \in \mathbb{N}, ptr = 0$          {Pointer which stores the length of the prefix of the state reported as judicial report
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
    recv (BreakAccProp, $verdict$, $toBreak$, $internalState$) from I/O:
        if $verdict$[public] ensures individual accountability)$\wedge$
          all other entries in $verdict$ map to $\varepsilon \wedge$
          $toBreak = \{$(consistency, public)$\}$:            {Handle violation of accountability w.r.t. consistency
            reply (BreakAccProp, true, $\varepsilon$)
        else:
            reply (BreakAssumption, false, $\varepsilon$)
    recv (BreakAssumption, $toBreak$, $internalState$) from I/O:   {Do not generate leakage when breaking assumptions
        reply (BreakAssumption, $\varepsilon$)
    recv (GetJudicialReport, $msg$, $internalState$) from I/O:                       {Generate judicial report
        send responsively (GetJudicialReport, $msg$, $internalState$) to NET  $(\star)$  {Forward request to $\mathcal{A}$ an wait for a pointer to the
                                                                                           state including an extension of this state
        wait for (GetJudicialReport, $ptr$, $ext$)
        if $ptr <$ maxHonestOutput $\vee$ $ptr > |state| \vee$
          $ext \not\subset \mathbb{N} \times \{0,1\}^* \vee$
          $ext\neg$ start with index $ptr + 1 \vee$                {Require valid input from $\mathcal{A}$, prefix must at
          further elements $\neg$ consecutive enumerated:           least contain honest output and extension
            go to $(\star)$                                           needs to be valid
        $ptr \leftarrow ptr$
        Let $state_{ptr}$ the prefix of the $state$ from the beginning including the $ptr$' component unite with $ext$.
        reply (GetJudicialReport, $state_{ptr}$)                                        {Return state as report
    recv (Corrupt, (public, $sid$, judge), $internalState$) from I/O:  {$\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$ declines corruption requests for the public judge
        reply false                                                           {The public judge is incorruptible
    recv (CorruptionStatus?(public, $sid$, judge), $internalState$) from I/O: {$\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$ asks for the public judge's corruption
                                                                                 status
        reply false                                                           {The public judge is incorruptible
```

Fig. 11: The judge parameter functionality $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$ for $\mathcal{F}^{\text{acc}}_{\text{cp}}$.

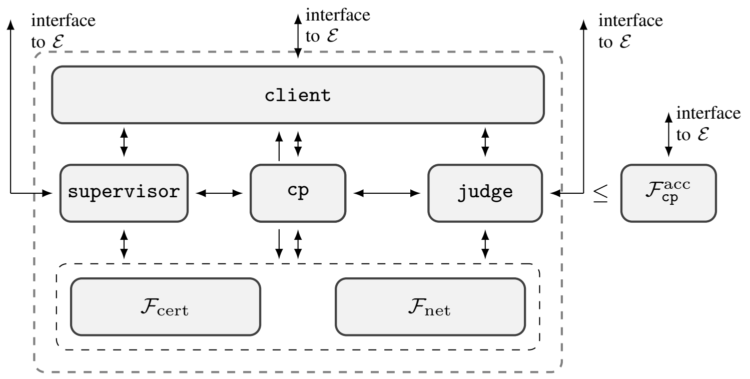

interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
client
supervisor
cp
judge
$\leq$
$\mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}$
$\mathcal{F}_{\mathrm{cert}}$
$\mathcal{F}_{\mathrm{net}}$

Fig. 12: Possible realization relation of the consensus service. The system $\mathcal{E}$ denotes the environment, modeling, as usual in UC setting, arbitrary higher level protocols. All machines are additionally connected to the adversary.

the prefix of the state that was already outputted to honest parties. As we model a public judge here, the judge is incorruptible. Thus, $\mathcal{F}_{\mathrm{judgeParams}}^{\mathrm{acc\text{-}cp}}$, answers `false` to `Corrupt` and `CorruptionStatus`? queries.

In what follows, we will assume that we already have a realization for $\mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}$ which we scale via the scaling protocol. Figure 12 gives an overview over a centralized consensus service CS (including relevant subroutines) in the client-server model which can be used to realize $\mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}$ and includes (public) accountability w.r.t. consistency and assumption-based liveness. Importantly, the consensus service CS has to sign answers to read request from its clients in this case. Thus, the (public) judge can gather undeniable evidence whether CS misbehaved according based on CS signatures and can ensure accountability w.r.t. consistency. As already explained in Section 3.1, communication via $\mathcal{F}_{\mathrm{net}}$ allows including breakable liveness guarantees. Of course, $\mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}$ can also be realized by a distributed system. For example, the accountable version of Apache Kafka [4] as presented in [50] is a good candidate to realize $\mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}$.

D.2 The Scaling Protocol: In this section, we provide some additional details regarding the consensus scaling protocol $\mathcal{P}_{\mathrm{cp}}^{\mathrm{acc}}$ we use in our example introduced in Section 3.1. We provide formal definitions for $\mathcal{P}_{\mathrm{cp}}^{\mathrm{acc}}$ in Figures 13 to 18.

Remark: To simplify presentation in Section 3.1, we introduced $\mathcal{P}_{\mathrm{cp}}^{\mathrm{acc}}$ as $(\mathtt{client}, \mathtt{supervisor}, \mathtt{judge} \mid \mathtt{scd}, \mathcal{F}_{\mathrm{net}}, \mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}}, \mathcal{F}_{\mathrm{cert}})$. Formally, the scaling protocol is defined as:

$$\mathcal{P}_{\mathrm{cp}}^{\mathrm{acc}} = (\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}} : \mathtt{client}, \mathcal{P}_{\mathrm{sv}}^{\mathrm{CD}} : \mathtt{supervisor}, \mathcal{P}_{\mathrm{judge}}^{\mathrm{CD}} : \mathtt{judge} \mid \mathcal{P}_{\mathrm{CD}} : \mathtt{scd}, \mathcal{F}_{\mathrm{net}} : \mathtt{net}, \mathcal{F}_{\mathrm{cp}}^{\mathrm{acc}} : \mathtt{client}, \mathcal{F}_{\mathrm{cert}}).$$

Client $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$. Clients of the scaling protocol $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ (cf. Figure 13) are the connection point between higher-level protocols/the environment and the scaling protocol. An instance of $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ models one party accessing the scaling protocol. The environment may queue messages/transactions via $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ for consensus and read (from) the established totally ordered sequence of transactions. If an instance $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ of $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ receives a `Submit` command, it firstly queries $\mathcal{A}$ which CD it should use to submit the transaction. Then, it assigns a unique ID to the message and signs $(id, tx, \mathsf{pid}_{\mathsf{cur}})$ via $\mathcal{F}_{\mathrm{cert}}$ and sends the submit request $(id, tx, \mathsf{pid}_{\mathsf{cur}}, \sigma)$ (where σ is the signature from above) via $\mathcal{F}_{\mathrm{net}}$ to the determined CD.

If the environment triggers a read request at an instance $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ of $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$, $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ first queries $\mathcal{A}$ which CD to use to process the request. Then, it forwards the request via $\mathcal{F}_{\mathrm{net}}$ to the CD. $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ also receives the response to the query via $\mathcal{F}_{\mathrm{net}}$. To pull the response from the CD from $\mathcal{F}_{\mathrm{net}}$, $\mathcal{A}$ triggers $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ with the `Pull` command. $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ then queries $\mathcal{F}_{\mathrm{net}}$ for new messages and verifies that the state in the delivered message is validly signed. In this case, $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$ forwards the message to the judge $\mathcal{P}_{\mathrm{judge}}^{\mathrm{CD}}$, waits for its reactivation, and then forwards the response to the environment.

The consensus distributors $\mathcal{P}_{\mathrm{CD}}$. Though the CD are the core of the scaling protocol, they are internal/private parties which use an internal/private consensus service to achieve consistency/consensus. Internal/private means that the interfaces of $\mathcal{P}_{\mathrm{CD}}$ are not accessible to higher-level protocols. An instance of $\mathcal{P}_{\mathrm{CD}}$ (cf. Figure 14) models one dedicated consensus distributor. Communication to/from the environment is always via $\mathcal{P}_{\mathrm{client}}^{\mathrm{CD}}$.

Messages from $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$ to $\mathcal{P}_{\mathrm{CD}}$ are sent via $\mathcal{F}_{\mathrm{net}}$ and vice versa. If $\mathcal{A}$ triggers $\mathcal{P}_{\mathrm{CD}}$ via the `Pull` message, $\mathcal{P}_{\mathrm{CD}}$ pulls for a new message at $\mathcal{F}_{\mathrm{net}}$. *(i)* If the pulled message contains a `Submit` command, $\mathcal{P}_{\mathrm{CD}}$ verifies the signature of the client submitting the transaction. If the signature verifies, it queues the submit request at $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$ for consensus. In particular, $\mathcal{P}_{\mathrm{CD}}$ forwards the request to its client component of $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$. *(ii)* If the pulled message is a `Read` request, $\mathcal{P}_{\mathrm{CD}}$ creates a signature over state and replies with the state including the created signature via $\mathcal{F}_{\mathrm{net}}$ to the requestor.

If $\mathcal{A}$ wants $\mathcal{P}_{\mathrm{CD}}$ to update its state, it sends `Update` to $\mathcal{P}_{\mathrm{CD}}$. Thereupon, $\mathcal{P}_{\mathrm{CD}}$ sends a `Read` request to $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$. $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$ replies with a `Read` message including a state update. $\mathcal{P}_{\mathrm{CD}}$ verifies that the state update has a valid format and that it is an extension of its current state. If both checks succeed, $\mathcal{P}_{\mathrm{CD}}$ overwrites its current state with the provided update.

The public judge $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$. The public judge $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ (cf. Figure 15) collects information *(i)* from clients $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$ and *(ii)* from the lower-level consensus service $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$. In contrast to clients which push evidence data to $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$, the judge pulls data from the lower-level public judge of $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$ on every activation during preprocessing. More specifically, as long as verdicts $= \varepsilon$, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ queries $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$'s judge for the most recent verdicts and a judicial report and overwrites its current (empty) verdicts and judicialReport with the received data. The judge only processes evidence in the case that there is no verdict so far.

If $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ receives evidence from $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$ which contains some CD's state (we assume that CD is identified with $(pid, sid, role)$) including its pid and a signature over the data. $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ checks that the signature verifies (which implicitly includes a check that pid is indeed a CD). If the signature does not verify, the reported evidence is ignored. Otherwise, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ checks that the reported state is a prefix of the most recent judicialReport. If this check fails, this means that $(pid, sid, role)$ violated consistency. Thus, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ will blame $(pid, sid, role)$ for its misbehavior.

On request `GetVerdict`, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ outputs the verdict stored in verdicts. If the environment invokes the `GetJudicialReport` interface, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ outputs the judicial report it gathered from $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$'s judge which is stored in judicialReport.

As $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ models a public judge, it also provides the `GetEvidence` interface to $\mathcal{A}$. $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ provides all collected evidence (W), the current judicial report, and its full transcript to $\mathcal{A}$ if $\mathcal{A}$ invokes this interface. This models that all data used by $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ is indeed public (as $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ makes it public). $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$'s `VerResult` interface further allows $\mathcal{A}$ to verify signatures of the parties of $\mathcal{P}^{\mathrm{acc}}_{\mathsf{cp}}$.

The public supervisor $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$. The supervisor $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$ (cf. Figure 16) is responsible for determining whether the assumption-based security property `liveness` still holds true and to make the corruption status of internal parties/parties from lower-level protocols accessible to the environment. If the environment queries $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$ via the (`IsAssumptionBroken`?, `liveness`, `public`), $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$ asks $\mathcal{F}_{\mathrm{net}}$ via `getDeliveryStatus` whether liveness still holds true and also queries $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$'s public supervisor and asks whether `liveness` still holds true for $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$. If both checks succeed, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$ responses that `liveness` still holds true for the scaling protocol. Otherwise, it reports the property as broken.

If the environment queries for the corruption status of internal parties of the protocol via `corruptInt`?, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{sv}}$ checks whether the matching instance of $\mathcal{P}_{\mathrm{CD}}$ is corrupted or whether the party at $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$, resp. the internal party at $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$ is corrupted and outputs the corruption status accordingly.

The network functionality $\mathcal{F}_{\mathrm{net}}$. The ideal network functionality $\mathcal{F}_{\mathrm{net}}$ (cf. Figure 18) with breakable liveness mainly provides a `Send` and a `Pull` interface to parties. There should be one instance of $\mathcal{F}_{\mathrm{net}}$ per sid modeling one separated network for a protocol. In order to measure whether liveness is broken, $\mathcal{F}_{\mathrm{net}}$ includes an in-build clock (see below). When an entity sends a message via $\mathcal{F}_{\mathrm{net}}$ to another entity, $\mathcal{F}_{\mathrm{net}}$ firstly buffers the message including a unique ID, sender, receiver, and the round $\mathcal{F}_{\mathrm{net}}$ received the message in buffer$_{msg}$. Additionally, the message (including all metadata) is leaked to $\mathcal{A}$.

In order to change the sequence of messages in buffer$_{msg}$, $\mathcal{A}$ is allowed to permute the messages in buffer$_{msg}$ via `Permute` analogously to the `Permute` command in $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cp}}$.

If a party $(pid, sid, role)$ wants to access its messages, it can query $\mathcal{F}_{\mathrm{net}}$ via `Pull`. In this case, $\mathcal{F}_{\mathrm{net}}$ looks for the message with the smallest ID addressed to $(pid, sid, role)$, deletes this message from buffer$_{msg}$, and forwards the message to $(pid, sid, role)$.

To increase the current time stored in round, $\mathcal{A}$ can use the `UpdateRound` command. In the case that liveness still holds true, i.e., inTime $=$ `true`, $\mathcal{F}_{\mathrm{net}}$ checks whether the round received of all messages in buffer$_{msg}$ is less than δ time units in the past (where δ is the liveness parameter of $\mathcal{F}_{\text{net}}$). If the check succeeds, $\mathcal{F}_{\text{net}}$ increases round by one. Otherwise, the round update is rejected. If liveness does not hold true anymore, $\mathcal{F}_{\text{net}}$ accepts all `UpdateRound` requests of $\mathcal{A}$.

Via `BreakLiveness`, $\mathcal{A}$ can break the liveness guarantees of $\mathcal{F}_{\text{net}}$. The other subroutines can query for the time of $\mathcal{F}_{\text{net}}$ via `GetCurRound` and they can also query via `getDeliveryStatus` whether liveness holds true or not. (Note that $\mathcal{F}_{\text{net}}$ is private in the considered protocol, thus not accessible to the environment).

The ideal signature functionality $\mathcal{F}_{\text{cert}}$ with in-built CA. In the scaling protocol model, we use $\mathcal{F}_{\text{cert}}$ (cf. Figure 17)– an adapted version of the standard ideal signature functionality $\mathcal{F}_{\text{sig}}$ (cf. Figure 31) as presented in [17] (which we use later on as well). The major difference between $\mathcal{F}_{\text{cert}}$ and $\mathcal{F}_{\text{sig}}$ is that $\mathcal{F}_{\text{cert}}$ includes an "in-build" CA. This simplifies the handling of signatures in this model.

This concludes the description of $\mathcal{P}^{\text{acc}}_{\text{cp}}$. For a description of the corruption behavior, we point to Section 3.1.

D.3 UC Security Analysis: We now present the full proof of Theorem 1 in detail. To ease notation, we use $\mathcal{F}^{\text{acc}}_{\text{cp}}$ for the ideal consensus service subroutine on the real side/hybrid setting and call the instance of the ideal consensus service we want to realize $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$.

Theorem 6. *Let $\delta \in \mathbb{N}$ be the upper time boundary for message delivery and $\Sigma = (\mathsf{gen}(1^\eta), \mathsf{sig}, \mathsf{ver})$ be an EUF-CMA secure signature scheme. Let $\mathcal{P}^{\text{acc}}_{\text{cp}}$ be as defined above. Let $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ as described above with $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$ as subroutine with parameters $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{consistency}\}$, $\mathsf{Sec}^{\mathsf{assumption}} = \{\texttt{liveness}\}$, $\mathsf{pids}_{\mathsf{judge}} = \{\texttt{public}\}$, and $\mathsf{ids}_{\mathsf{assumption}} = \{\texttt{public}\}$. Then:*

$$\mathcal{P}^{\text{acc}}_{\text{cp}} \leq (\mathcal{F}^{\text{acc}\prime}_{\text{cp}} \mid \mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}})$$

Proof. We firstly define a responsive simulator $\mathcal{S}$ such that the real world running the protocol $\mathcal{R} := \mathcal{P}^{\text{acc}}_{\text{cp}} = (\mathcal{P}^{\text{CD}}_{\text{client}} : \texttt{client}, \mathcal{P}^{\text{CD}}_{\text{sv}} : \texttt{supervisor}, \mathcal{P}^{\text{CD}}_{\text{judge}} : \texttt{judge} \mid \mathcal{P}_{\text{CD}} : \texttt{scd}, \mathcal{F}_{\text{net}} : \texttt{net}, \mathcal{F}^{\text{acc}}_{\text{cp}} : \text{client}, \mathcal{F}_{\text{cert}})$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$, with the protocol $\mathcal{I} := (\mathcal{F}^{\text{acc}}_{\text{cp}} \mid \mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}})$, for every ppt environment $\mathcal{E}$.

The simulator $\mathcal{S}$ is defined as follows: as common, $\mathcal{S}$ is a single machine. It is connected to $\mathcal{I}$ and the environment $\mathcal{E}$ via their network interfaces. In a run, there is only a single instance of the machine $\mathcal{S}$ that accepts and processes all incoming messages. The simulator $\mathcal{S}$ internally simulates the realization $\mathcal{R}$, including its behavior on the network interface connected to the environment, and uses this simulation to compute responses to incoming messages. For ease of presentation, we will refer to this internal simulation by $\mathcal{R}'$. More precisely, the simulation runs as follows:

Network communication from/to the environment

- Messages that $\mathcal{S}$ receives on the network connected to the environment (and which are hence meant for $\mathcal{R}$) are forwarded to the internal simulation $\mathcal{R}'$.
- Any messages sent by $\mathcal{R}'$ on its network interface (that are hence meant for the environment) are forwarded to the environment $\mathcal{E}$.

Corruption handling

- The simulator $\mathcal{S}$ keeps the corruption status of entities in $\mathcal{R}'$ and $\mathcal{I}$ synchronized. That is, whenever an entity of $\mathcal{P}^{\text{CD}}_{\text{client}}$, $\mathcal{P}_{\text{CD}}$, $\mathcal{F}^{\text{acc}}_{\text{cp}}$, or an internal entity of $\mathcal{F}^{\text{acc}}_{\text{cp}}$ in $\mathcal{R}'$ starts to consider itself corrupted, the simulator first corrupts the corresponding (internal) entity of $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ in $\mathcal{I}$ before continuing its simulation. Note that corruption of internal entities, i.e., of $\mathcal{P}_{\text{CD}}$, or (non-internal) parties of $\mathcal{F}^{\text{acc}}_{\text{cp}}$ is mapped to a corruption of an internal party in $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ in $\mathcal{I}$.
- Incoming Messages from corrupted (non-internal) entities of $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ in $\mathcal{I}$ are forwarded on the network to the environment in the name of the corresponding entity $\mathcal{P}^{\text{CD}}_{\text{client}}$ in $\mathcal{R}'$. Conversely, whenever a corrupted entity of $\mathcal{P}^{\text{CD}}_{\text{client}}$ wants to output a message to a higher-level protocol, $\mathcal{S}$ instructs the corresponding entity of $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ to output the same message to the higher-level protocol.
- For indirectly corrupted parties:[9] Note that these parties are (directly) corrupted in $\mathcal{I}$. For these instances, $\mathcal{I}$ forwards the inputs from $\mathcal{S}$. An indirectly corrupted party in $\mathcal{R}'$ produces an output on an I/O tape then $\mathcal{S}$ instructs $\mathcal{I}$ to forward this message to the intended receiver.

[9]Indirectly corrupted parties are not directly corrupted parties, i. e., $\mathcal{A}$ did not send $\mathsf{Corrupt}$ to the party so far, but (in our case) **DetermineCorrStatus** outputs true. That is, the party considers itself to be corrupted, e. g., due to a corrupted subrouting. For example, an instance of $\mathcal{P}^{\text{CD}}_{\text{client}}$ considers itself corrupted if its accompanied `signer` session at $\mathcal{F}_{\text{cert}}$ is corrupted.

Transaction submission
Whenever an honest entity $entity = (pid, sid, role)$ of $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ receives a request $(\texttt{Submit}, msg)$ to submit a new transaction msg, $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ buffers the message for later adding it to the state and leaks the full message including the ID of the message in $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$'s buffer to $\mathcal{S}$. $\mathcal{S}$ uses the leaked message to simulate the input of msg to $\mathcal{P}_{\text{client}}^{\text{CD}}$ in $\mathcal{R}'$.

In the case that a corrupted party submits a message (which can be extracted from $\mathcal{F}_{\text{net}}$), $\mathcal{S}$ submits the transaction via the Submit command to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$.

Read requests
Whenever an honest entity $(pid, sid, role)$ receives a request Read to read from the global state, $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ buffers this read request and leaks the request to $\mathcal{S}$ and waits for a trigger to process the read request. Upon receiving this request, $\mathcal{S}$ uses the leaked message to simulate the input of this message to $\mathcal{P}_{\text{client}}^{\text{CD}}$ in $\mathcal{R}'$ and stores *(i)* the ID provided by $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ for further processing and *(ii)* which entity triggered the read request.

If $\mathcal{A}$ triggers a simulated entity $(pid, sid, role)$ of $\mathcal{P}_{\text{client}}^{\text{CD}}$ to output the response to a read request via Pull, $\mathcal{S}$ simulates the input of Pull in $\mathcal{R}'$. If $(pid, sid, role)$ wants to output a state via a $(\texttt{Read}, state)$ message on its I/O connection in $\mathcal{R}'$, $\mathcal{S}$ extract the connected request ID id matching the request (see above) and sends $(\texttt{Deliver}, id)$ to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. If $\mathsf{brokenProps}[\texttt{consistency}, \texttt{public}] = \texttt{true}$ in $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$,[10] $\mathcal{S}$ simply forwards $(\texttt{Read}, state, (pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r))$ from $\mathcal{R}'$ including the stored receiver as response to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ Read request. Otherwise, $\mathcal{S}$ compares $state$ with state from $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$[11] and extracts the pointer N, such that state up to the N^{th} entry matches $state$. $\mathcal{S}$ then forwards $(\texttt{Read}, N, (pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r))$ to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ where $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r)$ is the recorded requestor of read request.

State updates
If there occurs a state update in $\mathcal{R}'$ in the simulation of $\mathcal{F}_{\text{cp}}^{\text{acc}}$, i.e., the variable state of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ is updated, $\mathcal{S}$ triggers a state update at $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. Therefore, it firstly retrieves the buffer of submitted messages/transactions which are not part of the state so far via getBuffer and the state$'$ from $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ via Read. It checks, which entries of state from the simulation are missing in state$'$ in $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. It then derives a permutation over the received buffer such that the entries missing in state$'$ have the same order as in state starting with the lowest ID in the buffer. Then, $\mathcal{S}$ sends this permutation via Permute to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ activates $\mathcal{S}$ again with an ack message. Thereupon, $\mathcal{S}$ sends Update to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ and is activated again via ack. $\mathcal{S}$ repeats sending the Update message until state and state$'$ are equal.

Public judge and supervisor

- If during the simulation of $\mathcal{P}_{\text{judge}}^{\text{CD}}$ in $\mathcal{R}'$, $\mathcal{P}_{\text{judge}}^{\text{CD}}$ renders a verdict, i.e., $\mathsf{verdicts} \neq \varepsilon$, $\mathcal{S}$ extracts the verdict and forwards it to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. More specifically, $\mathcal{S}$ sends $(\texttt{BreakAccProp}, verdict, \{(\texttt{consistency}, \texttt{public})\})$ to $(\texttt{public}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{cp}}^{\text{acc}} : \texttt{judge})$ where $verdict$ is a map from $\mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ where only the entry $verdict[\texttt{public}] = \mathsf{verdicts}$ (contains the verdict from $\mathcal{P}_{\text{judge}}^{\text{CD}}$) and all other entries are mapped to ε.
- If $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ is queried for a judicial report, it allows $\mathcal{S}$ to determine some details of the judicial report which is derived from $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$'s state. If $\mathcal{S}$ receives $(\texttt{GetJudicialReport}, msg, internalState)$ from $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$, it extracts the judicial report $report$ (as currently produced) from $\mathcal{F}_{\text{cp}}^{\text{acc}}$. It extracts the necessary pointer ptr to map $report$ to state from $\mathcal{F}_{\text{cp}}^{\text{acc}}$ and then sends $(\texttt{GetJudicialReport}, ptr, \varepsilon)$ back to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$.
- If $\mathcal{S}$ receives GetEvidence or VerResult via NET, it forwards this input to its simulated version of $\mathcal{P}_{\text{judge}}^{\text{CD}}$ and sends the output of the simulated machine back via NET. For the transcript provided as response to GetEvidence, $\mathcal{S}$ maps data such as if the simulated version of $\mathcal{P}_{\text{judge}}^{\text{CD}}$ would be directly connected to NET, etc.
- As soon as liveness breaks *(i)* in $\mathcal{F}_{\text{net}}$ or *(ii)* in $\mathcal{F}_{\text{cp}}^{\text{acc}}$ in $\mathcal{R}'$, $\mathcal{S}$ breaks liveness at $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$. In the case of *(i)*, as soon as $\mathsf{inTime} = \texttt{false}$ in $\mathcal{F}_{\text{net}}$, $\mathcal{S}$ sends $(\texttt{BreakAssumption}, \{(\texttt{liveness}, \texttt{public})\})$ to $(\texttt{public}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{cp}}^{\text{acc}} : \texttt{supervisor})$. In Case *(ii)*, if the $\mathsf{brokenProps}[\texttt{liveness}, \texttt{public}] = \texttt{true}$ at $\mathcal{F}_{\text{cp}}^{\text{acc}}$, $\mathcal{S}$ sends the same message as above to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$.

Further details

- $\mathcal{S}$ keeps the clocks/rounds of $\mathcal{R}'$ and $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ synchronous. That is, $\mathcal{S}$ sends UpdateRound to $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$ whenever a round update in the simulated $\mathcal{F}_{\text{net}}$ is performed and before continuing the simulation.

This concludes the description of the simulator. It is easy to see that *(i)* $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded [12]

[10]Note that $\mathcal{S}$ explicitly breaks this property. Thus, $\mathcal{S}$ knows the value of $\mathsf{brokenProps}[\texttt{consistency}, \texttt{public}]$ in $\mathcal{F}_{\text{cp}}^{\text{acc}\prime}$.

[11]This is possible, as $\mathcal{S}$ determines the state via Update commands.

[12]This is the polynomial runtime notion employed by the iUC framework.

and *(ii)* $\mathcal{S}$ is a responsive simulator for $\mathcal{I}$, i.e., restricting messages from $\mathcal{I}$ are answered immediately as long as $\{\mathcal{S},\mathcal{I}\}$ runs with a responsive environment. We now argue that $\mathcal{R}$ and $\{\mathcal{S},\mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$.

Let $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$ be an arbitrary but fixed environment. First, observe that $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ provides $\mathcal{S}$ with full information about all submit and read requests performed by higher-level protocols. Hence, the simulated protocol $\mathcal{R}'$ within $\mathcal{S}$ obtains the same inputs and thus performs identical to the real world $\mathcal{R}$. Therefore, the network behavior simulated by $\mathcal{S}$ towards the environment is indistinguishable from the network behavior of $\mathcal{R}$. Note that state changes triggered via network interface are synchronized between $\mathcal{R}'$ and $\mathcal{I}$. Together with the state synchronization during I/O interaction (see below), the simulator can keep the states of $\mathcal{R}'$ and $\mathcal{I}$ in synchronization. Moreover, we can also conclude that the corruption status of (internal) entities in the real and ideal world is synchronized. Since the simulator has full control over corrupted entities, which are handled via the internal simulation $\mathcal{R}'$, this implies that the I/O behavior of corrupted entities of $\mathcal{R}/\mathcal{I}$ towards higher-level protocols/the environment is also identical in the real and ideal world. Note that purely internal (private) parties have no interface to the environment. Thus, the only way to potentially distinguish the real and ideal world is the I/O behavior of honest entities of $\mathcal{R}/\mathcal{I}$ towards higher-level protocols.

We will now go over all possible interactions with honest entities on the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the environment, i.e., are also indistinguishable. At the start of a run, there were no interactions on the I/O interface with honest parties yet. In the following, assume that all I/O interactions so far have resulted in the same behavior visible towards the environment in both the real and ideal world.

Submission requests: Submission requests do not directly result into an output to the environment. But, they might affect the output of $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ at a later point in time as they have direct impact of the state of $\mathcal{F}_{\text{cp}}^{\text{acc}'}$, resp. $\mathcal{R}$. Thus, we now show that submit requests behave "identical", i.e., we have to argue that these changes are "synchronized" between $\mathcal{I}$ and $\mathcal{R}'$. In particular, the buffered set of transactions in $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ is a subset of the buffered set of transactions in $\mathcal{R}'$ (which are buffered *(i)* as `Submit` message in $\mathcal{F}_{\text{net}}$ and *(ii)* in the buffer of $\mathcal{F}_{\text{cp}}^{\text{acc}}$).

Observe that, upon receiving a submission request, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ behaves similar to $\mathcal{R}$: in $\mathcal{F}_{\text{cp}}^{\text{acc}'}$, the submitted transaction is directly stored into the buffer. In $\mathcal{R}'$, the submit message is forwarded via $\mathcal{F}_{\text{net}}$ to $\mathcal{P}_{\text{CD}}$. So, the "buffers" of $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ and $\mathcal{R}'$ stay synchronized when defining the buffer of $\mathcal{R}'$ as the set of `Submit` messages in $\mathcal{F}_{\text{net}}$'s buffer union the buffer of $\mathcal{F}_{\text{cp}}^{\text{acc}}$. Note that as soon as submit request of corrupted parties are visible to $\mathcal{S}$ (as specified above), $\mathcal{S}$ also submits these transactions to $\mathcal{F}_{\text{cp}}^{\text{acc}'}$. Thus, $\mathcal{S}$ is indeed able to keep the buffers in $\mathcal{I}$ and $\mathcal{R}$ synchronized.

State updates: As buffers in $\mathcal{R}'$ and $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ are synchronized, we can conclude that the states of $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ and $\mathcal{R}'$ stay synchronized during updates as this boils down to mirror the state from $\mathcal{F}_{\text{cp}}^{\text{acc}}$ to $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ (as $\mathcal{S}$ does). *(i)* The buffer of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ is a subset of the buffer of $\mathcal{F}_{\text{cp}}^{\text{acc}'}$. *(ii)* Submitted transaction which are not in the intersection of the states of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ and $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ are currently in the buffer of $\mathcal{F}_{\text{net}}$. *(iii)* By construction, both provide the same security guarantees. Thus, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ provides at most less guarantees than $\mathcal{F}_{\text{cp}}^{\text{acc}}$ (in case that properties are broken at $\mathcal{F}_{\text{cp}}^{\text{acc}'}$). Thus, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ will accept every state update of $\mathcal{S}$ and both, $\mathcal{I}$ and $\mathcal{R}'$ stay synchronized. Note that $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ also provides $\mathcal{S}$ the possibility to forward answers to read requests which have not been requested. As this is possible in real and ideal world, both worlds stay indistinguishable.

Read requests: Observe that, upon receiving a read request, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ stores the read request in a buffer and leaks the full request to the simulator including an ID and the receiver of the response. Note that the procedure is analogously to a submission requests. A read request does not directly result into an output to the environment. With the same argumentation as above follows that $\mathcal{S}$ is able to keep the "read buffers" of $\mathcal{I}$ and $\mathcal{R}'$ synchronized.

Deliver response to read request: If $\mathcal{S}$ is triggered to output a response to a read request (via `Pull`), there are two cases to consider: *(i)* $\mathsf{brokenProps}[\texttt{consistency}, \texttt{public}] = \texttt{true}$ or *(ii)* $\mathsf{brokenProps}[\texttt{consistency}, \texttt{public}] = \texttt{false}$. In the first case, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ will simply forward the input from $\mathcal{S}$ to the requestor. As the output is extracted from $\mathcal{R}'$ we can conclude that $\mathcal{R}$ and $\mathcal{I}$ are indistinguishable. In the second case, $\mathcal{S}$ can extract the response from $\mathcal{R}'$. Honest clients in $\mathcal{R}'$ receive a validly signed prefix of the state of $\mathcal{F}_{\text{cp}}^{\text{acc}}$ (which is in synchronization with $\mathcal{F}_{\text{cp}}^{\text{acc}'}$'s state). Thus, $\mathcal{F}_{\text{cp}}^{\text{acc}'}$ will accept the `Deliver` command of $\mathcal{S}$ and the outputs in both worlds is identical. Thus, both worlds are indistinguishable.

Verdicts: Firstly, note that $\mathcal{S}$ sends a `BreakAccProp` message to break public consistency when consistency in $\mathcal{R}'$ breaks, resp. when there is a new verdict available in $\mathcal{R}'$s public judge.[13] Secondly, note that this request matches the rules in $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$: the verdict $\mathcal{S}$ provides ensures individual accountability and $\mathcal{S}$ solely tries to break public consistency. Thus, $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ accepts the verdict $\mathcal{S}$ provides. Further note that we assume the used signature scheme to be EUF-CMA secure. Thus, the probability that an honest party receives and accepts a state update that includes a forged signature (and forwards this to a judge) is negligible. Further, as `verifier` is incorruptible, Thus, we can conclude that $\mathcal{S}$ can always keep the verdicts and whether consistency is broken synchronized in $\mathcal{I}$ and $\mathcal{R}'$ and the output on a `GetVerdict` request is equal in $\mathcal{R}$ and $\mathcal{I}$. Thus, both worlds are/stay indistinguishable.

Judicial Reports: Observe that the states of $\mathcal{F}^{\text{acc}}_{\text{cp}}$ in $\mathcal{R}'$ and $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ are synchronized (see above). As the judicial report is derived as a prefix of the state, $\mathcal{S}$ always provides a pointer to the full state of both $\mathcal{R}'$ and $\mathcal{I}$ to determine the judicial report of $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$. Thus, delivered judicial reports are equal in both worlds and $\mathcal{R}$ and $\mathcal{I}$ stay indistinguishable.

`GetEvidence` **and** `VerResult`**:** We note that the environment cannot use the `GetEvidence` or the `VerResult` interface of $\mathcal{P}^{\text{CD}}_{\text{judge}}$ to distinguish between real and ideal world. The activation order of the simulated $\mathcal{P}^{\text{CD}}_{\text{judge}}$ matches the real world order and also messages to $\mathcal{P}^{\text{CD}}_{\text{judge}}$ match in both worlds perfectly. As $\mathcal{S}$ maps the transcript such that it matches the real world execution the transcript (and provided state) is indistinguishable in both worlds. Similarly, $\mathcal{S}$ fully simulates all interaction with $\mathcal{F}_{\text{cert}}$ (in the same order in both worlds). Thus, requests in both worlds will lead to the same output.

Supervisor: The same holds true for `BreakAssumption` requests to the supervisor. The message of $\mathcal{S}$ matches the rules and thus will be accepted. This allows $\mathcal{S}$ to keep both worlds synchronized regarding the assumption-based security property liveness. Thus, both worlds will answer requests to the supervisor identically, i. e., both worlds remain indistinguishable.

Time requests: As the simulator updates the internal clock of $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ every time $\mathcal{F}_{\text{net}}$'s clock in $\mathcal{R}'$ is increased, both worlds always output the same value for the current time. Note that, even if $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ enforces *liveness*, any round update requests of $\mathcal{S}$ will indeed be accepted: as $\mathcal{S}$ keeps the liveness guarantees of $\mathcal{I}$ and $\mathcal{R}'$ synchronized, $\mathcal{S}$ will not ask for a time update that violates the liveness check in $\mathcal{F}^{\text{acc}\prime}_{\text{cp}}$ as long as liveness still holds true.

We also note that the (additional) public evidence $\mathcal{A}$, resp. $\mathcal{E}$, may gather at the $\mathcal{P}^{\text{CD}}_{\text{judge}}$ (via `GetEvidence`) does not enable $\mathcal{E}$ to distinguish between both worlds. As $\mathcal{R}'$ gets the same input as $\mathcal{I}/\mathcal{R}$, the output of `GetEvidence` in $\mathcal{R}'$ is the same as in the real world $\mathcal{R}$. We emphasize that the judicial report of $\mathcal{P}^{\text{CD}}_{\text{judge}}$ already includes most of the information $\mathcal{E}$ receives via `GetEvidence`. The transcript is the same in $\mathcal{I}$ and $\mathcal{R}$ as the order of activations of and the inputs to $\mathcal{P}^{\text{CD}}_{\text{judge}}$, resp. its simulated version, are identically. Additionally, `GetEvidence` provides signatures to $\mathcal{E}$. However, as signatures do not contain private information and are ideally generated in both worlds, it is not possible to distinguish between $\mathcal{R}$ and $\mathcal{I}$ based on the additional signature data.

Altogether, $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ behave identical in terms of behavior visible to the environment $\mathcal{E}$ and thus are indistinguishable.

$\square$

D.4 Deterrence Analysis: Common consensus services, e. g., Hashgraph [11], charge fees for their service. Clients also benefit from a consensus service as they do not have to establish a consensus service on their own. Thus, we conclude $U^i_{hP} > 0$ for all parties involved in the consensus service. As $\mathcal{P}^{\text{acc}}_{\text{cp}}$ provides public accountability and verdicts are fair (by definition), we can conclude that the public judge will not falsely accuse an honest party. Further, all data used in the judging procedure is public. Thus, we can assume that $U^i_{hE} = U^i_{hL} = 0$. To deter parties from misbehavior, penalties for violating consistency need to be sufficiently high. Thus, we assume that $U^i_{mL} \gg 0$. In particular, penalties need to negate potential profit due to malicious behavior, i. e., it needs to hold true $U^i_{mP} - U^i_{mL} < U^i_{hP}$. The exact value of U^i_{mP} and hence the minimally required fees depend on the context that uses the consensus protocol. As the judging procedure would identify faked evidence, we assume $U^i_{mE} = 0$. Thus, Equation 1 and 2 hold true.

[13]We remark that $\mathcal{P}^{\text{CD}}_{\text{judge}}$ does not update her verdict.

Description of the client $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}} = (\texttt{client})$ of the consensus distributors:

```
Participating roles: {client}
Corruption model: Dynamic corruption without secure erasures
```

Description of $M_{\texttt{client}}$:

```
Implemented role(s): {client}
Subroutines: $\mathcal{F}_{\mathrm{cert}}$ : verifier, $\mathcal{F}_{\mathrm{cert}}$ : signer, $\mathcal{F}_{\mathrm{net}}$ : net, $\mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ : judge
Internal state:
    – owner $\in (\{0,1\}^*)^3$        {The entity $(pid, sid, role)$ that has access to this instance of $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$
    – serial $\in \mathbb{N}$, serial $= 0$        {The serial for "agreed" transactions
CheckID($pid, sid, role$):
    Accept all messages for the same $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$.
Corruption behavior:
    DetermineCorrStatus($pid, sid, role$):
        if corrupted = true:        {Checks whether a party itself is corrupted.
            return true
        $corrRes \leftarrow$ corr($\mathsf{pid_{cur}}, \mathsf{sid_{cur}}$, signer)        {Request corruption status at $\mathcal{F}_{\mathrm{cert}}$
        return $corrRes$        {Return whether $\mathcal{F}_{\mathrm{cert}}$ instance is corrupted
Initialization:
        owner $\leftarrow (\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$        {Record that only $(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$ has access to this instance of $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$
Main:
        recv (Submit, $tx$) from I/O s.t. $(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$ = owner:        {Transaction submission with specified CD
            send responsively (Submit, $tx$) to (NET)        {$\mathcal{A}$ desides which CD $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$ will use
            wait for (Submit, $tx$, $pid_{\mathrm{CD}}$)
            serial $\leftarrow$ serial $+ 1$
            $\sigma' \leftarrow$ sign([$serial$, $tx$, $\mathsf{pid_{cur}}$])
            send (Send, $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$, $(pid_{\mathrm{CD}}, \mathsf{sid_{cur}}, \mathsf{scd})$, (Submit, serial, $tx$, $\mathsf{pid_{cur}}$, $\sigma$)) to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\mathrm{net}}$ : net)
                                                                        {Send transaction via $\mathcal{F}_{\mathrm{net}}$ to $\mathcal{P}_{\mathrm{CD}}$
        recv Read from I/O s.t. $(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$ = owner:        {Read request from IO
            send responsively Read to (NET)        {$\mathcal{A}$ desides which CD $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$ will use
            wait for (Read, $pid_{\mathrm{CD}}$)
            send (Send, $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$, $(pid_{\mathrm{CD}}, \mathsf{sid_{cur}}, \mathsf{scd})$, (Read, $\mathsf{pid_{cur}}$)) to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\mathrm{net}}$ : net)        {Forward request via $\mathcal{F}_{\mathrm{net}}$
        recv Pull from NET:        {$\mathcal{A}$ triggers pulling at $\mathcal{F}_{\mathrm{net}}$
            send Pull to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\mathrm{net}}$ : net)
            wait for (Pull, $msg$)
            if $msg$ = (Read, $state$, $pid_{\mathrm{CD}}$, $\sigma$):
                $b \leftarrow$ verifySig($pid_{\mathrm{CD}}$, $state$, $\sigma$)
                if $b \wedge$ state $\subset state$:
                    state $\leftarrow state$
                    send (Evidence, $pid_{\mathrm{CD}}$, $state$, $\sigma$) to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{P}^{\mathrm{CD}}_{\mathrm{judge}}$ : judge)
                    wait for ack
                    send (Read, $state$) to owner        {Output response of read request to owner
Procedures and Functions:
        function sign($msg$) :        {Sign message at $\mathcal{F}_{\mathrm{cert}}$
            send (sign, $msg$) to $(\mathsf{pid_{cur}}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{cert}}$ : signer)
            wait for (sign, $\sigma$)
            return $\sigma$
        function verifySig($pid$, $msg$, $\sigma$) :        {Verify signature at $\mathcal{F}_{\mathrm{cert}}$
            send (VerResult, $msg$, $\sigma$) to $(\mathsf{pid_{cur}}, (pid, \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{cert}}$ : verifier)
            wait for (VerResult, $result$)
            return $result$
```

Fig. 13: The consensus distributors client $\mathcal{P}^{\mathrm{CD}}_{\mathrm{client}}$.

Description of the consensus distributors $\mathcal{P}_{\text{CD}} = (\texttt{scd})$:

```
Participating roles: {scd}
Corruption model: Dynamic corruption without secure erasures
```

Description of $M_{\texttt{scd}}$:

```
Implemented role(s): {scd}
Subroutines: $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : client, $\mathcal{F}_{\text{cert}}$ : verifier, $\mathcal{F}_{\text{cert}}$ : signer, $\mathcal{F}_{\text{net}}$ : net, $\mathcal{P}_{\text{judge}}^{\text{CD}}$ : judge
Internal state:
    – state $\subset \mathbb{N} \times \{0,1\}^*$, state $= \emptyset$          {The set of totally ordered transactions/messages
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Corruption behavior:
    DetermineCorrStatus($pid, sid, role$):
        if corrupted = true:                                   {Checks whether a party itself is corrupted.
            return true
        $corrRes_1 \leftarrow$ corr(pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{cert}}$ : signer)        {Request corruption status at $\mathcal{F}_{\text{cert}}$
        $corrRes_2 \leftarrow$ corr(pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : client)     {Request corruption status at $\mathcal{F}_{\text{cp}}^{\text{acc}}$
        return $corrRes_1 \lor corrRes_2$            {Return whether $\mathcal{F}_{\text{cert}}$ or $\mathcal{F}_{\text{cp}}^{\text{acc}}$ instance is corrupted
Main:
    recv Pull from NET:                                  {$\mathcal{A}$ triggers pid$_{\text{cur}}$ to query $\mathcal{F}_{\text{net}}$
        send Pull to (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{net}}$ : net)
        wait for (Pull, $msg$)
        if $msg$ = (Submit, $serial, tx, pid, \sigma$):              {Received message is a Submit
            $b \leftarrow$ verifySig($pid$, ($serial, tx, pid$), $\sigma$)          {Check valid signature
            if $b$:                          {If signature is valid, submit data to consensus service
                send (Submit, ($serial, tx, pid, \sigma$)) to (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : client)   {Forward tx to consensus service
        else if $msg$ = (Read, $pid$):                              {In case of a Read request
            $\sigma \leftarrow$ sign(state)                                 {Sign current state
            send (Send, (pid$_{\text{cur}}$, sid$_{\text{cur}}$, role$_{\text{cur}}$), ($pid$, sid$_{\text{cur}}$, client), (Read, state, pid$_{\text{cur}}$, $\sigma$) to (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{net}}$ : net)
                                                    {Reply with full state including signature
    recv Update from NET:                              {Request update at consensus service
        send Read to (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : client)       {Forward request to consensus service
    recv (Read, $state$) from (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : client):      {State update from consensus service
        require: $state$ is a set with entries of form ($ctr, msg$),
                 where $ctr \in \mathbb{N}$, $msg \in \{0,1\}^*$.
        if state $\subset$ $state$:
            state $\leftarrow$ $state$
Procedures and Functions:
    function sign($msg$) :                                  {Sign message at $\mathcal{F}_{\text{cert}}$
        send (sign, $msg$) to (pid$_{\text{cur}}$, (pid$_{\text{cur}}$, sid$_{\text{cur}}$), $\mathcal{F}_{\text{cert}}$ : signer)
        wait for (sign, $\sigma$)
        return $\sigma$
    function verifySig($pid, msg, \sigma$) :                     {Verify signature at $\mathcal{F}_{\text{cert}}$
        send (VerResult, $msg, \sigma$) to (pid$_{\text{cur}}$, ($pid$, sid$_{\text{cur}}$), $\mathcal{F}_{\text{cert}}$ : verifier)
        wait for (VerResult, $result$)
        return $result$
```

Fig. 14: The consensus distributor ITM $\mathcal{P}_{\text{CD}}$ which scales consensus.

E. Accountable Key Exchange based on an accountable PKI (Full Details)

In this section, we provide full details regarding the accountable PKI case study as presented in Section 3.2. This section is structured as follows: we firstly present the details regarding the accountable PKI as introduced in Section 3. In particular, we provide a detailed introduction to the ideal accountable PKI functionality $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. Afterwards, we explain the protocol $\mathcal{P}_{\text{PKI}}^{\text{acc}}$ and provide a formal specification of the model. We conclude this part with a security analysis of the accountable PKI.

E.1 An Accountable Ideal PKI Functionality $\mathcal{F}_{\text{PKI}}^{\text{acc}}$: We now present the full specification of the ideal accountable PKI functionality $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ including the formal specification of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ in Figure 19 including its subroutine $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$ in Figure 20. $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ extends Canetti et al.'s $\mathcal{G}_{\text{BB}}$ as presented in [27].

One session of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ models one instance of a PKI including several different CAs and CTLs. From a high-level perspective, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ offers the possibility to register certificates for identities at a dedicated CA and to retrieve certificates of identities from CAs. We model that parties use a local PKI client which is responsible for the communication with the PKI, i. e., the client is connected via I/O to the environment modeling higher-level protocols which access/use the PKI. To interact with the PKI, honest parties, resp. clients identified by $(pid, sid, \texttt{client})$, can register a certificate for their *own* identity. To register a certificate, clients send a message of the form $(\texttt{Register}, pk, pid_{\texttt{CA}})$ (where $pid_{\texttt{CA}}$ is the PID of the CA that should attest the entities identity and pk be a string, typically the public key of the client) to $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ does not immediately process the

Description of $\mathcal{P}^{\text{CD}}_{\text{judge}} = (\texttt{judge})$:

Participating roles: $\{\texttt{judge}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{judge}}$:

```
Implemented role(s): {judge}
Subroutines: $\mathcal{F}_{\text{cert}}$ : verifier, $\mathcal{F}^{\text{acc}}_{\text{cp}}$ : judge
Internal state:
  – counter $\in \mathbb{N}$, counter $= 0$                                   {Evidence counter
  – W $\subset \mathbb{N} \times (\{0,1\}^*)^2$     {Reported evidence (enumerated) per party, entries of form $(ctr, pid, state)$
  – judicialReport $\in \{0,1\}^*$, judicialReport $= \varepsilon$        {The judicial report from $\mathcal{F}^{\text{acc}}_{\text{cp}}$
  – verdicts $\in \{0,1\}^*$, verdicts $= \varepsilon$                          {Recorded verdict
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$ addressed to $(\texttt{public}, sid, \texttt{judge})$.
MessagePreprocessing:
    if verdicts $= \varepsilon$:
        send GetVerdict to $(\texttt{public}, \mathsf{sid_{cur}}, \mathcal{F}^{\text{acc}}_{\text{cp}} : \texttt{judge})$     {Query for verdicts regarding consensus service
        wait for (GetVerdict, $verdict$)
        verdicts $\leftarrow verdict$
    if verdicts $= \varepsilon$:                                                    {Update judicial report
        send GetJudicialReport to $(\texttt{public}, \mathsf{sid_{cur}}, \mathcal{F}^{\text{acc}}_{\text{cp}} : \texttt{judge})$
        wait for (GetJudicialReport, $judicialReport$)
        judicialReport $\leftarrow judicialReport$
Main:
    recv (Evidence, $pid_{\text{CD}}, state, \sigma$) from I/O:                  {Evidence from clients
        $b \leftarrow$ verifySig($pid_{\text{CD}}, state, \sigma$)
        if $b$:
            if $state \not\subset$ judicialReport:   {Honest CD will always forward state wich is a subset of a judicial report
                verdicts.add(dis($pid_{\text{CD}}$, $\mathsf{sid_{cur}}$, $\mathcal{P}_{\text{CD}}$ : scd))
        reply ack
    recv GetVerdict from I/O:
        reply (GetVerdict, verdicts)
    recv (GetJudicialReport, $msg$) from I/O:        {Return judicial report (from lower-level) to requestor
        reply (GetJudicialReport, judicialReport)
    recv GetEvidence from NET:                        {$\mathcal{A}$ may query the public judge for the
        reply (GetEvidence, W, judicialReport, transcript[a])   evidence it gathered including all details
    recv VerResult($pid, msg, \sigma$) from NET:      {$\mathcal{A}$ verify signatures via $\mathcal{P}^{\text{CD}}_{\text{judge}}$'s interface to $\mathcal{F}_{\text{cert}}$
        $result \leftarrow$ verifySig($pid, msg, \sigma$
        reply (VerResult, $result$)
Procedures and Functions:
    function verifySig($pid, msg, \sigma$) :                                    {Verify signature at $\mathcal{F}_{\text{cert}}$
        send (VerResult, $msg, \sigma$) to ($\mathsf{pid_{cur}}$, ($pid$, $\mathsf{sid_{cur}}$), $\mathcal{F}_{\text{cert}}$ : verifier)
        wait for (VerResult, $result$)
        return $result$
```

[a]transcript is a special variable in iUC, which, informally speaking, contains a transcript of all messages sent and received by the current machine instance.

Fig. 15: The judging functionality $\mathcal{P}^{\text{CD}}_{\text{judge}}$ for $\mathcal{P}^{\text{acc}}_{\text{cp}}$ (the consensus scaling protocol).

request, it stores the register message including the pid of the party which requests the certificate in a buffer $\mathsf{buffer_S}$ for later processing. This allows to model asynchronous network. Also, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ leaks the full request to $\mathcal{A}$. If an honest party $(pid_r, sid, \texttt{client})$ wants to access a certificate of a party pid, the process is similar: the party sends $(\texttt{Retrieve}, pid, pid_{\texttt{CA}})$ to access the certificate of identity pid at CA $pid_{\texttt{CA}}$. Retrieve requests are also stored in a buffer $\mathsf{buffer_R}$ including the requestor $(pid_r, sid, \texttt{client})$ for processing later on. $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ also leaks the full request to $\mathcal{A}$.

As $\mathcal{A}$ has full control over the network, $\mathcal{A}$ needs to inform $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ when it should accept a certificate and add it to its state. Therefore, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ allows $\mathcal{A}$: *(i)* to drop entries from $\mathsf{buffer_S}$ and *(ii)* to add buffered certificates to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$'s state. In Case *(i)*, $\mathcal{A}$ sends a $(\texttt{Drop}_S, pid, pk, pid_{\texttt{CA}})$ message to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. If the entry $(pid, pk, pid_{\texttt{CA}})$ exists in $\mathsf{buffer_S}$, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ removes the entry from $\mathsf{buffer_S}$ and replies $(\texttt{Drop}_S, \texttt{ack})$ to $\mathcal{A}$. Otherwise, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ declines the request and responds $(\texttt{Drop}_S, \texttt{nack})$ to $\mathcal{A}$. In Case *(ii)*, $\mathcal{A}$ sends the command $(\texttt{Update}, buffer)$ to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$, where $buffer$ is expected to be a subset of $\mathsf{buffer_S}$. If $buffer \subset \mathsf{buffer_S}$, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ adds the elements from $buffer$ to the state and removes them from the $\mathsf{buffer_S}$. $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ then replies to $\mathcal{A}$ that it accepted the update.

Similarly, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ handles retrieve requests. $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ allows the adversary $\mathcal{A}$ *(i)* to drop entries from $\mathsf{buffer_R}$ and

Description of $\mathcal{P}_{\text{sv}}^{\text{CD}} = (\texttt{supervisor})$:

```
Participating roles: {supervisor}
Corruption model:    incorruptible
```

Description of $M_{\texttt{supervisor}}$:

```
Implemented role(s): {supervisor}
Subroutines: $\mathcal{F}_{\text{cp}}^{\text{acc}}$ : supervisor, $\mathcal{F}_{\text{net}}$ : net
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
    recv (IsAssumptionBroken?, $prop$, $id$):
        if $id$ = public ∧ $prop$ = liveness:        {The model includes public liveness as assumption-based security property
            send getDeliveryStatus to ($\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\text{net}}$ : net)        {Query $\mathcal{F}_{\text{net}}$ whether liveness still holds
            wait for (getDeliveryStatus, $inTime$)
            send (IsAssumptionBroken?, liveness, public) to ($\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\text{cp}}^{\text{acc}}$ : supervisor)    {Query $\mathcal{F}_{\text{cp}}^{\text{acc}}$'s supervisor whether liveness still holds
            wait for (IsAssumptionBroken?, $b$)
            if $inTime \wedge \neg b$:        {Liveness holds true
                reply (IsAssumptionBroken?, false)
            else:
                reply (IsAssumptionBroken?, true)
        else:
            reply (IsAssumptionBroken?, false)
    recv (corruptInt?, ($pid, sid, role$)) from I/O s.t. $role \notin$ {client, judge, supervisor}:
        $corrRes$ ← false; $b$ ← false        {Check corruption depending on protocol layer
        if $role$ = scd:        {Handle internal parties of $\mathcal{P}_{\text{cp}}^{\text{acc}}$
            $corrRes$ ← corr(($pid, sid, role$))        {Check whether party is corrupted if directly accessible
        else:        {Handle internal parties from lower protocol levels
            send (corruptInt, ($pid, sid, role$)) to (public, $\mathsf{sid_{cur}}, \mathcal{F}_{\text{cp}}^{\text{acc}}$ : supervisor)  {Check if internal party at $\mathcal{F}_{\text{cp}}^{\text{acc}}$ is corrupted
            wait for (corruptInt, $b$)
        if $corrRes \vee b$:
            reply (corruptInt, true)
        else:
            reply (corruptInt, false)
    recv (BreakAssumption, $toBreak$, $internalState$) from I/O:        {Leakage request
        reply (BreakAssumption, $\varepsilon$)        {Provide empty leakage
```

Fig. 16: The supervisor $\mathcal{P}_{\text{sv}}^{\text{CD}}$ for $\mathcal{P}_{\text{CS}}$.

(ii) to trigger the response to a retrieve request. Case *(i)* is analogously to Case *(i)* from the paragraph above. Here, $\mathcal{A}$ sends a $(\texttt{Drop}_R, pid, pid_{\texttt{CA}}, pid_r, role_r)$ message to $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. If the entry $(pid, pid_{\texttt{CA}}, pid_r, role_r)$ exists in $\mathsf{buffer_R}$, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ removes the entry from $\mathsf{buffer_R}$ and replies $(\texttt{Drop}_R, \texttt{ack})$ to $\mathcal{A}$. Otherwise, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ declines the request and responds $(\texttt{Drop}_R, \texttt{nack})$ to $\mathcal{A}$. In Case *(ii)*, $\mathcal{A}$ sends $(\texttt{Deliver}, pid, pid_{\texttt{CA}}, pid_r, role_r, pk_{\mathcal{A}})$ to $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. There are now two cases to consider: *(i)* $\texttt{correctCert}$ is broken for $(pid, sid, \texttt{client})$ or the party is corrupted and *(ii)* $\texttt{correctCert}$ is not broken for $(pid, sid, \texttt{client})$ and the party is not corrupted. In the latter case, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ removes the entry from $\mathsf{buffer_R}$ and sends the requested output (extracted from its local state) via $(\texttt{Retrieve}, pid, pid_{\texttt{CA}}, \mathsf{state}[pid, pid_{\texttt{CA}}])$ to $(pid_r, sid, role_r)$ (and ignores further data provided by $\mathcal{A}$). Otherwise, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ allows $\mathcal{A}$ to determine pid's certificate. That is, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ forwards the public key $pk_{\mathcal{A}}$ provided by $\mathcal{A}$ via $(\texttt{Retrieve}, pid, pid_{\texttt{CA}}, pk_{\mathcal{A}})$ to $(pid_r, sid, role_r)$.

To cover accountability properties, we expect $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ to be used with the parameters $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{correctCert}\}$ and $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$. For $\texttt{correctCert}$, we require verdicts which state that one or several CAs misbehaved. $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ includes local judges – one local judge $((\texttt{local}, pid, \texttt{client}), sid, \texttt{judge})$ per client identity $(pid, sid, \texttt{client})$, i.e., $\mathsf{pids_{judge}}$ is expected to be $\{\texttt{local}\} \times \{0,1\}^* \times \{\texttt{client}\}$. As judicial report $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$ outputs an empty report. Further, the functionality $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$ specifies that verdicts need to be of form $\bigwedge_{i=1}^{n} \mathsf{dis}(pid_{\texttt{CA}}^{i}, \mathsf{sid_{cur}}, \texttt{ca}), n \in \mathbb{N}$, where $(pid_{\texttt{CA}}^{i}, \mathsf{sid_{cur}}, \texttt{ca})$ are CA identities in $\mathcal{P}_{\text{PKI}}^{\text{acc}}$. As the model does not include assumption-based security properties, we expect $\mathsf{ids_{assumption}} = \emptyset$ and specify that $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$ always outputs ε as response to a $\texttt{BreakAssumption}$ request. Note that *(i)* local judges act as pure message forwarder for $\mathcal{A}$ if the associated party is corrupted and *(ii)* $\mathcal{A}$ may corrupt a local judge only if its accompanied party, resp. client is already corrupted.

E.2 An Accountable PKI based on CTLs: In what follows, we provide the full specification of our PKI protocol, resp. its model $\mathcal{P}_{\text{PKI}}^{\text{acc}}$, as presented in Section 3. We also provide a formal description of all components

Description of the protocol $\mathcal{F}_{\text{cert}} = (\texttt{signer}, \texttt{verifier})$:

```
Participating roles: {signer, verifier}
Corruption model: incorruptible                                              {See text below
Protocol parameters:
    – $\mathsf{p} \in \mathbb{Z}[x]$.        {Polynomial that bounds the runtime of the algorithms provided by the adversary.
    – $\eta \in \mathbb{N}$                  {The security parameter.
    – sig   {Signing algorithm, outputs a signature $\sigma$ on input $(msg, \mathsf{sk})$. The generated signature has a length of $\eta$ bits
    – ver   {Signature verifying algorithm, outputs verification result on input $(msg, \sigma, \mathsf{pk})$
    – gen   {Key generation algorithm, outputs $(\mathsf{pk}, \mathsf{sk})$ on input $1^\eta$
```

Description of $M_{\texttt{signer},\texttt{verifier}}$:

```
Implemented role(s): {signer, verifier}
Internal state:
    – $(\mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$.          {Key pair.
    – $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$.                               {Party ID of the key owner.
    – $\mathsf{msglist} \subset \{0,1\}^* = \emptyset$.                                      {Set of recorded messages.
    – $\mathtt{corrupted} \in \{\mathtt{true}, \mathtt{false}\} = \mathtt{false}$.          {Is signature key corrupted?
CheckID($pid, sid, role$):
    Check that $sid = (pid', sid')$:
    If this check fails, output reject.           {A single instance manages all parties and roles in a
    Otherwise, accept all entities with the same SID.   single session. A session models one signature key pair
Corruption behavior:                                    belonging to party $pid'$.
    – DetermineCorrStatus($pid, sid, role$): Return corrupted.
Initialization:
        $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\eta)$                     {Generate public/secret key pair
        Parse $\mathsf{sid_{cur}}$ as $(pid, sid)$.
        $\mathsf{pidowner} \leftarrow pid$.
Main:
        recv (Sign, $msg$) from I/O to (pidowner, _, signer):
            $\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
            add $msg$ to msglist.
            reply (Signature, $\sigma$).                 {Record $msg$ for verification and return signature.
        recv (Verify, $msg, \sigma$) from I/O to (_, _, verifier):
            $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$.                {Verify signature.
            if $b = \mathtt{true} \land msg \notin \mathsf{msglist} \land \mathtt{corrupted} = \mathtt{false}$:
                reply (VerResult, false).                {Prevent forgery.
            else:
                reply (VerResult, $b$).                  {Return verification result.

        recv corruptSigKey from NET:                     {Allow network attacker to corrupt signature keys.
            corrupted ← true.
            reply (corruptSigKey, ok).
```

Fig. 17: The ideal signature functionality $\mathcal{F}_{\text{cert}}$.

of the protocol in Figure 22 to 31. Figure 21 provides an overview over the different components of the PKI protocol.

Remark: The full formal definition of $\mathcal{P}^{\text{acc}}_{\text{PKI}}$ as presented in Section 3.1 is $\mathcal{P}^{\text{acc}}_{\text{PKI}} = (\mathcal{P}^{\mathtt{CA}}_{\text{client}} : \mathtt{client}, \mathcal{P}^{\mathtt{CA}}_{\text{sv}} : \mathtt{supervisor}, \mathcal{P}^{\mathtt{CA}}_{\text{judge}} : \mathtt{judge} \mid \mathcal{P}^{\mathtt{CA}} : \mathtt{ca}, \mathcal{P}_{\mathtt{CTL}} : \mathtt{ctl}, \mathcal{F}_{\text{psync-net}} : \mathtt{psync\text{-}net}, \mathcal{F}_{\text{sig}}, \mathcal{F}_{\text{auth}} : \mathtt{auth})$.
We remark that one PKI consisting of clients, several CAs, and several CTLs is modeled as one instance (sid) of $\mathcal{P}^{\text{acc}}_{\text{PKI}}$.

CA clients $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$. The CA client $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ (cf. Figure 22 and 23) handles the connection between the PKI and the environment. $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$'s main purpose is to request at most one certificate at the PKI and to query for certificates of other parties. Additionally, $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ also monitors the PKI, resp. its CTLs and surveils whether there exists solely certificates for its identity requested by itself.

If the environment instructs an instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$, namely the instance for $(pid, sid, \mathtt{client})$, to register a string pk as its public key at a certain CA $(pid_{\mathtt{CA}}, \mathsf{sid_{cur}}, \mathtt{ca})$, it sends $(\mathtt{Register}, pk, pid_{\mathtt{CA}})$ via an authenticated channel $\mathcal{F}_{\text{auth}}$ to $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$. An honest instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ registers at most one certificate. If the environment calls the register command once again, $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ does not process the request.

If the environment wants to retrieve a certificate of pid issued by $pid_{\mathtt{CA}}$, it instructs its instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ to do so via $(\mathtt{Retrieve}, pid, pid_{\mathtt{CA}})$. $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ stores the request in requests and forwards the request via NET, i. e., unprotected, to $(pid_{\mathtt{CA}}, sid, \mathtt{ca})$ (with the same session ID). Note that there is neither a guarantee that the request arrives at the CA nor that it arrives unchanged. When $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ receives the answer to the retrieve

Description of the protocol $\mathcal{F}_{\text{net}} = (\texttt{net})$:

```
Participating roles: {net}
Corruption model: incorruptible
Protocol parameters:
    – $\delta \in \mathbb{N}$                                   {The expected liveness guarantee for $\mathcal{F}_{\text{net}}$
```

Description of $M_{\texttt{net}}$:

```
Implemented role(s): {net}
Internal state:
    – $\mathsf{buffer}_{msg} \subset \mathbb{N}^2 \times (\{0,1\}^*)^7$, $\mathsf{buffer}_{msg} = \emptyset$   {Buffer for messages consisting of tuples $(ctr, round, sender, receiver, content)$
    – $\mathsf{counter} \in \mathbb{N}$, $\mathsf{counter} = 0$                    {Counter for messages
    – $\mathsf{round} \in \mathbb{N}$, $\mathsf{round} = 0$                        {Current round/time unit
    – $\mathsf{inTime} \in \{\texttt{true}, \texttt{false}\}$, $\mathsf{inTime} = \texttt{true}$   {Indicator whether messages where within $\delta$ time units
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
    recv $(\texttt{Send}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), (pid, \mathsf{sid_{cur}}, role), msg)$ from I/O: {$(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$ sends a message to $(pid, \mathsf{sid_{cur}}, role)$
        $\mathsf{counter} \leftarrow \mathsf{counter} + 1$
        $\mathsf{buffer}_{msg}.\mathsf{add}([\mathsf{counter}, \mathsf{round}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), (pid, \mathsf{sid_{cur}}, role), msg])$   {Record message
        send $(\texttt{Send}, \mathsf{counter}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), (pid, \mathsf{sid_{cur}}, role), msg)$ to NET   {Forward leakage and identifier to $\mathcal{A}$
    recv Pull from I/O:                                          {Parties can pull messages
        Let $(counter, (pid, \mathsf{sid_{cur}}, role), (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), msg) \in \mathsf{buffer}_{msg}$, such that there exists no smaller $counter$
        with recipient $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})$.
        if $(counter, (pid, \mathsf{sid_{cur}}, role), (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), msg)$ as above exists:
            $\mathsf{buffer}_{msg}.\texttt{remove}([counter, (pid, \mathsf{sid_{cur}}, role), (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), msg])$   {Remove message from delivery queue
            reply $(\texttt{Pull}, msg)$                                {Deliver message to receiver
        else:
            reply $(\texttt{Pull}, \bot)$
        recv $(\texttt{Permute}, \pi)$ from NET:                   {$\mathcal{A}$ is allowed to change the order in the $\mathsf{buffer}_{msg}$
            require: $\pi$ is a permutation of the IDs in $\mathsf{buffer}_{msg}$.
            for all $(ctr, msg) \in \mathsf{buffer}_{msg}$ where $msg$ represents all other content of $\mathsf{buffer}_{msg}$ besides $ctr$ do:
                $\mathsf{buffer}_{msg}.\texttt{remove}([ctr, msg])$
                $\mathsf{buffer}_{msg}.\mathsf{add}([\pi(ctr), msg])$
    recv UpdateRound from NET:                                   {$\mathcal{A}$ triggers round update
        if $\exists(\_, round, \_, \_) \in \mathsf{buffer}_{msg}$, s.t. $\mathsf{round} - round > \delta \ \wedge \mathsf{inTime} = \texttt{true}$:
            reply $(\texttt{UpdateRound}, \texttt{false}, \epsilon)$
        else:
            $\mathsf{round} \leftarrow \mathsf{round} + 1$
            reply $(\texttt{UpdateRound}, \texttt{true}, \epsilon)$
        reply $(\texttt{UpdateRound}, \texttt{true}, \epsilon)$
    recv $(\texttt{GetCurRound})$:                              {$\mathcal{A}$ and $\mathcal{E}$ are allowed to query the current round.
        reply $(\texttt{GetCurRound}, \mathsf{round})$
    recv BreakLiveness from NET:                                 {$\mathcal{A}$ breaks liveness
        $\mathsf{inTime} \leftarrow \texttt{false}$
    recv getDeliveryStatus from I/O:                             {$\mathcal{E}$ may query whether messages where delivered "in time"
        reply $(\texttt{getDeliveryStatus}, \mathsf{inTime})$
```

Fig. 18: The simplified network model $\mathcal{F}_{\text{net}}$ with breakable δ-liveness.

request via the message $(\texttt{Retrieve}, (ctr, (serial, pid, pk, pid_{\texttt{CA}}), pid_{\texttt{CTL}}, \tau, \sigma))$ on the NET interface, $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ firstly request the current time τ' at $\mathcal{F}_{\text{psync-net}}$. Then, $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ updates its information regarding the CA $pid_{\texttt{CA}}$ and queries $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ (see below) for the public key of $pid_{\texttt{CA}}$ and the CTLs that the CAs uses including their public keys (this is done via the message $(\texttt{GetCaCtls}, pid_{\texttt{CA}})$). When $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ receives the answer $(\texttt{GetCaCtls}, pk_{\texttt{CA}}, (pid_1, \ldots, pid_n), (pk_1, \ldots, pk_n))$, it stores the CA's public key $pk_{\texttt{CA}}$ in $\mathsf{pk}_{\texttt{CA}}[pid_{\texttt{CA}}]$. Further, it stores the CTLs connected to the CA in $\mathsf{CTLs}[pid_{\texttt{CA}}]$ and their public keys in $\mathsf{pk}_{\texttt{CTL}}[pid_{\texttt{CTL}}]$. After $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ gathered the trusted information from $\mathcal{F}^{\texttt{CA}}_{\text{init}}$, it checks whether *(i)* $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ indeed requested the certificate for pid at $pid_{\texttt{CA}}$, *(ii)* the provided certificate is older than $3 \cdot \delta$ time units, *(iii)* $pid_{\texttt{CTL}}$ is a CTL for $pid_{\texttt{CA}}$, *(iv)* the certificate is validly signed by the CTL $pid_{\texttt{CTL}}$, and *(v)* the certificate is also validly signed by the CA $pid_{\texttt{CA}}$. If the checks succeed, $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ removes the request from its request queue and sends the certificate to the requestor.

The adversary is responsible for starting the certificate monitoring process of $\mathcal{P}^{\texttt{CA}}_{\text{client}}$.[14] When $\mathcal{A}$ triggers the monitoring process by sending $\texttt{Monitor}$, $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ updates the set of CTLs at $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ (by using the $\texttt{GetCTLs}$). Let CTL be the set of all CTLs provided by $\mathcal{F}^{\texttt{CA}}_{\text{init}}$. $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ then queries all CTLs via $(\texttt{Monitor}, \mathsf{pid_{cur}}, CTL)$ via $\mathcal{F}_{\text{psync-net}}$. Note that $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ and $\mathcal{F}_{\text{psync-net}}$ are connected via a (direct and thus secure) I/O interface. When

[14]We note that security guarantees are incorporated into $\mathcal{F}_{\text{psync-net}}$ which we will discuss later on.

Description of the ideal and accountable PKI $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}} = (\texttt{client}, \texttt{judge}, \texttt{supervisor})$:

```
Participating roles: $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Corruption model: dynamic without secure erasures
Protocol parameters:
    – $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$                                   {Accountability properties
    – $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$                            {Assumption-based security properties
    – $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$   {set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)
    – $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$                 {set of entities/IDs where properties are ensured via assumptions
```

Description of $M^{\mathrm{acc\text{-}PKI}}_{\texttt{client}}$:

```
Implemented role(s): $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Subroutines: $\mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams}$
Internal state:
    – $\mathsf{state} : \{0,1\}^* \times \{0,1\}^* \rightarrow, \mathsf{state} = \emptyset$    {The registered certificates, resp. identities (as mapping), initially $\perp$ for all entries
    – $\mathsf{buffer}_{\mathsf{S}} \subset (\{0,1\}^*)^3, \mathsf{buffer}_{\mathsf{S}} = \varepsilon, \mathsf{buffer}_{\mathsf{R}} \subset (\{0,1\}^*)^4, \mathsf{buffer}_{\mathsf{R}} = \emptyset$ {Requested certificates/buffer of form $(pid, pid_{\mathtt{CA}}, pid_r, role_r)$
    – $\mathsf{corruptedIntParties} \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$ {The set of corrupted internal parties $(pid, sid, role)$
    – $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \rightarrow \{\texttt{true}, \texttt{false}\}$ {Stores broken security assumptions per id, initially false $\forall$entries
    – $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \rightarrow \{\texttt{true}, \texttt{false}\}$ {Stores broken security properties per judge/id, initially false $\forall$entries
    – $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \rightarrow \{0,1\}^*$      {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Corruption behavior:
    – AllowCorruption($pid, sid, role$):
        Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
        if $role = \texttt{judge}$:
            send $(\texttt{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
              to $(pid, sid, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams})$    {$\mathcal{F}_{\mathrm{judgeParams}}$ decides whether judges can be corrupted
            wait for $b$; return $b$
    – DetermineCorrStatus($pid, sid, role$):
        if $role = \texttt{judge}$:          {$\mathcal{F}_{\mathrm{judgeParams}}$ may determine a judge's corruption status
            send $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
              to $(pid, sid, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams})$
            wait for $b$; return $b$
    – AllowAdvMessage($pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m$)
        Do not allow sending messages to $\mathcal{F}_{\mathrm{judgeParams}}$.   {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}_{\mathrm{judgeParams}}$ in the name of corrupted parties.
MessagePreprocessing:
    if message is addressed to $((\texttt{local}, pid, role), sid, \texttt{judge})$ and $(pid, sid, role)$ is corrupted:
        Forward message to $\mathcal{A}$               {Forward request to corrupted local judges to $\mathcal{A}$
    if Receive $(\texttt{Fwd}, msg, (pid', sid', role'))$ to $((\texttt{local}, pid, role), sid, \texttt{judge})$ via NET and $(pid, sid, role)$ is corrupted:
        send $msg$ to $(pid, sid, role)$          {$\mathcal{A}$ corrupted local judge acts as message forwarder for $\mathcal{A}$
Main:
    recv $(\texttt{Register}, pk, pid_{\mathtt{CA}})$ from I/O:                         {Register certificate.
        $\mathsf{buffer}_{\mathsf{S}}.\mathsf{add}([\mathsf{pid}_{\mathsf{cur}}, pk, pid_{\mathtt{CA}}])$                  {Record registration attempt
        send $(\texttt{Register}, pk, \mathsf{pid}_{\mathsf{cur}}, pid_{\mathtt{CA}})$ to NET                 {Leak request to $\mathcal{A}$

    recv $(\texttt{Retrieve}, pid, pid_{\mathtt{CA}})$ from I/O:                        {Query for certificate
        $\mathsf{buffer}_{\mathsf{R}}.\mathsf{add}([pid, pid_{\mathtt{CA}}, \mathsf{pid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}])$
        send $(\texttt{Retrieve}, pid, pid_{\mathtt{CA}}, \mathsf{pid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ to NET       {Leak request to $\mathcal{A}$

    recv $(\texttt{Drop}_S, pid, pk, pid_{\mathtt{CA}})$ from NET:      {$\mathcal{A}$ is allowed to drop the register requests
        if $(pid, pk, pid_{\mathtt{CA}}) \in \mathsf{buffer}_{\mathsf{S}}$: $\mathsf{buffer}_{\mathsf{S}}.\mathtt{remove}([pid, pk, pid_{\mathtt{CA}}])$
            reply $(\texttt{Drop}_S, \texttt{ack})$
        else:
            reply $(\texttt{Drop}_S, \texttt{nack})$
                                                     {$\mathcal{A}$ may trigger when retrieve requests are answered.
    recv $(\texttt{Deliver}, pid, pid_{\mathtt{CA}}, pid_r, role_r, pk_{\mathcal{A}})$ from NET:    If correctCert is broken, $\mathcal{A}$ may inject malicious certificates.
        require: $(pid, pid_{\mathtt{CA}}, pid_r, role_r) \in \mathsf{buffer}_{\mathsf{R}}$
        if $\mathsf{brokenProps}[\texttt{correctCert}, (pid, \mathsf{sid}_{\mathsf{cur}}, \texttt{client})] = \texttt{false} \wedge (pid, \mathsf{sid}_{\mathsf{cur}}, \texttt{client}) \notin \mathsf{CorruptionSet}$:
            $\mathsf{buffer}_{\mathsf{R}}.\mathtt{remove}([pid, pid_{\mathtt{CA}}, pid_r, role_r])$
            send $(\texttt{Retrieve}, pid, pid_{\mathtt{CA}}, \mathsf{state}[pid, pid_{\mathtt{CA}}])$ to $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r)$
        else:
            $\mathsf{buffer}_{\mathsf{R}}.\mathtt{remove}([pid, pid_{\mathtt{CA}}, pid_r, role_r])$     {$\mathcal{A}$ determines the certificate if correctCert is broken
            send $(\texttt{Retrieve}, pid, pid_{\mathtt{CA}}, pk_{\mathcal{A}})$ to $(pid_r, \mathsf{sid}_{\mathsf{cur}}, role_r)$    for pid or pid is corrupted
    recv $(\texttt{Drop}_R, pid, pid_{\mathtt{CA}}, pid_r, role_r)$ from NET:       {$\mathcal{A}$ is allowed to drop the read requests
        if $(pid, pid_{\mathtt{CA}}, pid_r, role_r) \in \mathsf{buffer}_{\mathsf{R}}$:
            $\mathsf{buffer}_{\mathsf{R}}.\mathtt{remove}([pid, pid_{\mathtt{CA}}, pid_r, role_r])$
            reply $(\texttt{Drop}_R, \texttt{ack})$
        else:
            reply $(\texttt{Drop}_R, \texttt{nack})$

    recv $(\texttt{Update}, buffer)$ from NET s.t. $buffer \subset \mathsf{buffer}_{\mathsf{S}}$:   {$\mathcal{A}$ may trigger when $\mathsf{buffer}_{\mathsf{S}}$ is included in state
        for all $(pid, pk, pid_{\mathtt{CA}}) \in buffer$ do:
            if $\mathsf{state}[pid, pid_{\mathtt{CA}}] = \perp$:
                $\mathsf{state}[pid, pid_{\mathtt{CA}}] \leftarrow pk$
        $\mathsf{buffer}_{\mathsf{S}}.\mathtt{remove}[buffer]$                                  {Update buffer
        reply $(\texttt{Update}, \texttt{ack})$
    Include static code from the AUC transformation $\mathcal{T}_1(\cdot)$ here, i.e., include additional code from Figure 2 and 3 here.
  ${}^{a}\mathsf{Roles}_{\mathcal{F}} = \{\texttt{client}\}$ here.
```

Fig. 19: An ideal and accountable PKI $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ derived from $\mathcal{G}_{\mathrm{BB}}$ [27].

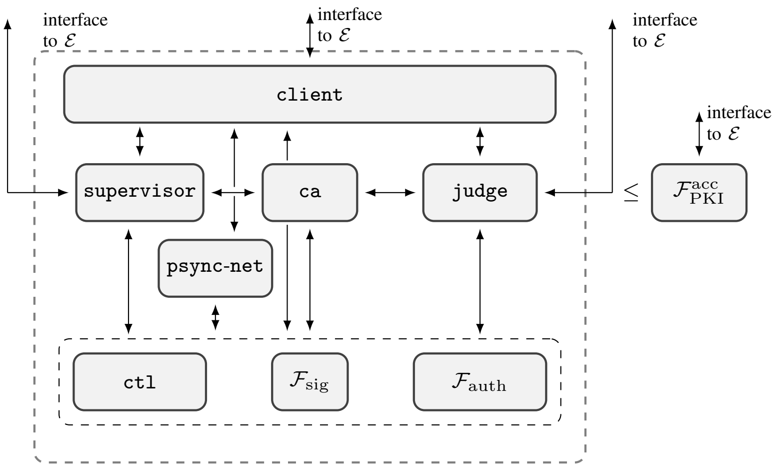
Description of $\mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}} = (\texttt{judgeParams})$:

```
Participating roles: {judgeParams}
Corruption model: incorruptible
```

Description of $M^{\text{acc-PKI}}_{\texttt{judgeParams}}$:

```
Implemented role(s): {judgeParams}
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
    recv ($\texttt{BreakAccProp}, verdict, toBreak, internalState$) from I/O:        {No restrictions on verdict
      if All entries in $verdict$ are of form $\bigwedge_{i=1}^{n} \mathsf{dis}(pid^i_{\mathtt{CA}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{ca})$, $n \in \mathbb{N}$ and all $pid^i_{\mathtt{CA}} \in \{0,1\}^*$:
        if $toBreak \subset \{\texttt{correctCert}\} \times (\{0,1\}^*)^3$:
          reply ($\texttt{Cheat}, \texttt{true}, \varepsilon$)
        else:
          reply ($\texttt{Cheat}, \texttt{false}, \varepsilon$)
      else:
        reply ($\texttt{Cheat}, \texttt{false}, \varepsilon$)
    recv ($\texttt{BreakAssumption}, toBreak, internalState$) from I/O:        {Do not generate leakage when breaking assumptions
      reply ($\texttt{BreakAssumption}, \varepsilon$)
    recv ($\texttt{GetJudicialReport}, msg, internalState$) from I/O:        {Generate judicial report
      reply ($\texttt{GetJudicialReport}, \varepsilon$)        {Return an empty report
    recv ($\mathsf{Corrupt}, (id, sid, \mathsf{judge}), internalState$) from I/O:        {$\mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}$ allows to corrupt a local judge iff the accompanied client is corrupted
      if $id = (\texttt{local}, pid, \texttt{client}) \wedge (pid, sid, \texttt{client}) \in \mathsf{CorruptionSet}$:
        reply true
      else:
        reply false
    recv ($\texttt{CorruptionStatus?}, (id, sid, \mathsf{judge}), internalState$) from I/O:        {$\mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}$ interprets a local judge as corrupted iff the accompanied client is corrupted
      if $(id, sid, \mathsf{judge}) \in \mathsf{CorruptionSet}$:
        reply true
      else if $id = (\texttt{local}, pid, \texttt{client}) \wedge (pid, sid, \texttt{client}) \in \mathsf{CorruptionSet}$ :
        reply true
      else:
        reply false
```

Fig. 20: $\mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}$ for $\mathcal{F}^{\text{acc}}_{\text{PKI}}$.



interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
client
interface to $\mathcal{E}$
supervisor
ca
judge
$\leq$
$\mathcal{F}^{\text{acc}}_{\text{PKI}}$
psync-net
ctl
$\mathcal{F}_{\text{sig}}$
$\mathcal{F}_{\text{auth}}$

Fig. 21: Realization relation of a CA/CTL example stated in Theorem 2. The system $\mathcal{E}$ denotes the environment, modeling, as usual in UC setting, arbitrary higher level protocols. All machines are additionally connected to the network adversary who is responsible for delivering network messages.

$\mathcal{F}_{\text{psync-net}}$ returns an answer of a CTL (containing all certificates issued for the requested pid recorded at that CTL), $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ pulls updates public keys and data of CAs and CTLs at $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ (see above). Afterwards, it verifies whether the received certificates are validly signed (by a CA and a CTL). If one of the certificates does not match the public key pk stored in $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$, $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ reports the falsely issued certificate to its local judge $\mathcal{P}^{\mathtt{CA}}_{\text{judge}}$. *Corruption:* The adversary can dynamically corrupt every instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$. In this case, $\mathcal{A}$ has full control

over the input and output interface of the instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ and can act on its behalf. Further note that an instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ considers itself corrupted if its session at $\mathcal{F}_{\text{sig}}$ is corrupted or if the used authenticated channel session at $\mathcal{F}_{\text{auth}}$ is corrupted.
Further note that:

- Every new instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ registers itself automatically at $\mathcal{F}_{\text{psync-net}}$ when activated for the first time.
- $\mathcal{A}$ is responsible to trigger an instance of $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ to register itself for the usage of authenticated channels (via $\mathcal{F}_{\text{auth}}$).

Remark: Note that we do not model that clients provide "self-signed certificates" as part of their certificate requests to a CA, i. e., they do sign their public key in combination with their identity (and potentially some metadata) and forward this certificate to the CA. We omit self-signed certificates in our model due to simplicity. To incorporate this into our model, one can extend $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ with a parameter (which is intended to include the signature scheme) for creating a self-signed certificate. Before requesting a certificate, $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$ then needs to invoke this parameter with the input data and has to enrich the certificate request with the output produced by the parameter. To cover this case, one also needs to adapt $\mathcal{P}^{\mathtt{CA}}$. It needs to be enhanced with a new parameter to check the validity of the client's self-signed certificate. $\mathcal{P}^{\mathtt{CA}}$ only process certificate request if the parameter (on input of the certificate request) accepts the certificate request.

Certification authorities $\mathcal{P}^{\mathtt{CA}}$. The main purpose of CAs, resp. $\mathcal{P}^{\mathtt{CA}}$ (cf. Figure 24), is to issue and to distribute certificates. There exists one instance of $\mathcal{P}^{\mathtt{CA}}$ per CA. The issuance of certificates is divided into two steps: *(i)* a CA generates a so-called pre-certificate and forwards the pre-certificate to a CTL. *(ii)* The CTL attests the correct generation of the (pre-)certificate and provides a finalized certificate to the CA. More specifically, $\mathcal{P}^{\mathtt{CA}}$ works as follows: On the first activation of an instance of $\mathcal{P}^{\mathtt{CA}}$, it queries the $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ for initialization. $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ initializes $\mathcal{P}^{\mathtt{CA}}$'s instance at $\mathcal{F}_{\text{sig}}$, i. e., $\mathcal{P}^{\mathtt{CA}}$'s signing and public key, and provides its public key pk to $\mathcal{P}^{\mathtt{CA}}$. $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ also informs $\mathcal{P}^{\mathtt{CA}}$ which CTLs it will/has to use (including their public keys).

As clients send their certificate requests via an authenticated channel $\mathcal{F}_{\text{auth}}$, $\mathcal{A}$ is responsible for triggering $\mathcal{P}^{\mathtt{CA}}$ to pull certificate requests from $\mathcal{F}_{\text{auth}}$. When $\mathcal{P}^{\mathtt{CA}}$ receives such request (of form $(\mathtt{Register}, pid, pk)$), it creates a pre-certificate: it adds an ID/serial number (an incremental counter) and its own identity to (pid, pk), and signs $(id, (pid, pk), \mathsf{pid}_{\mathsf{cur}})$. It then forwards the so-called pre-certificate $(id, (pid, pk), \sigma_{\mathtt{CA}})$ for finalization via NET to its connected CTLs. Note that $\mathcal{P}^{\text{acc}}_{\text{PKI}}$ does not enforce liveness properties here, i. e., we do not enforce that these requests are actually delivered to a CTL.

As $\mathcal{A}$ has full control over the network, we model that $\mathcal{A}$ triggers $\mathcal{P}^{\mathtt{CA}}$'s update process. If $\mathcal{P}^{\mathtt{CA}}$ receives a $(\mathtt{Read}, pid_{\mathsf{CTL}}, state, \sigma_{\mathsf{CTL}})$ message via NET from one of its CTLs (as stored in CTLs), $\mathcal{P}^{\mathtt{CA}}$ checks whether *(i)* the message is validly signed by a CTL pid_{CTL} from CTLs, *(ii)* the provided $state$ is an extension of the state previously delivered by pid_{CTL} (which is stored in $\mathsf{certs}[pid_{\mathsf{CTL}}]$), and *(iii)* all certificates in $state$ are validly signed by pid_{CTL}. If these checks succeed, $\mathcal{P}^{\mathtt{CA}}$ stores $state$ (i. e., the certificates provided by pid_{CTL}) as update in $\mathsf{certs}[pid_{\mathsf{CTL}}]$.

When clients want to retrieve a certificate from $\mathcal{P}_{\text{CS}}$, they send $(\mathtt{Retrieve}, pid, pid_{\mathtt{CA}})$ to the instance of $\mathcal{P}^{\mathtt{CA}}$, namely $(pid_{\mathtt{CA}}, sid, \mathtt{ca})$. Note that these requests are delivered via $\mathcal{A}$/NET and can thus be tampered. When receiving such a request, $\mathcal{P}^{\mathtt{CA}}$ checks in its storage certs whether it issued a pre-certificate for pid. If it issued a pre-certificate, $\mathcal{P}^{\mathtt{CA}}$ looks for the finalized certificate in the certificates provided by the CTLs. If $\mathcal{P}^{\mathtt{CA}}$ can find a finalized certificate in $\mathsf{certs}_{\mathsf{CTL}}$ (for any CTL), it forwards the first certificate matching the search criteria (via NET) to the requesting party.

Corruption: Analogously to $\mathcal{P}^{\mathtt{CA}}_{\text{client}}$, $\mathcal{A}$ can dynamically corrupt every instance of $\mathcal{P}^{\mathtt{CA}}$. In case of a corruption, $\mathcal{A}$ gains full control over the input and output interface of the instance of $\mathcal{P}^{\mathtt{CA}}$ and can act on its behalf. Further note that an instance of $\mathcal{P}^{\mathtt{CA}}$ considers itself corrupted if its session at $\mathcal{F}_{\text{sig}}$ is corrupted or if the used authenticated channel session at $\mathcal{F}_{\text{auth}}$ is corrupted.

Moreover, $\mathcal{A}$ is responsible for triggering an instance of $\mathcal{P}^{\mathtt{CA}}$ to register itself for the usage of authenticated channels (via $\mathcal{F}_{\text{auth}}$).

Certificate Transaction Logs $\mathcal{P}_{\mathsf{CTL}}$. CTLs validate and finalize (pre-)certificates. Further, they allow clients to check whether there exist falsely/maliciously issued certificates for their identities. We provide the specification of $\mathcal{P}_{\mathsf{CTL}}$ in Figure 25. One instance of $\mathcal{P}_{\mathsf{CTL}}$ models one CTL. Similarly to $\mathcal{P}^{\mathtt{CA}}$, $\mathcal{P}_{\mathsf{CTL}}$ queries the initialization functionality $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ for initialization on its first activation. $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ initializes $\mathcal{P}_{\mathsf{CTL}}$ instance at $\mathcal{F}_{\text{sig}}$ and provides its public key pk to $\mathcal{P}_{\mathsf{CTL}}$. $\mathcal{F}^{\mathtt{CA}}_{\text{init}}$ also informs $\mathcal{P}_{\mathsf{CTL}}$ for which CAs it will finalize pre-certificates.

If an instance of $\mathcal{P}_{\texttt{CTL}}$ identified by $(pid_{\texttt{CTL}}, sid, \texttt{ctl})$ receives a `Submit` request to finalize a pre-certificate (via NET), $\mathcal{P}_{\texttt{CTL}}$ checks whether the pre-certificate is of form $(id, (pid_r, pk_r), pid_{\texttt{CA}}, \sigma_{\texttt{CA}})$. If the current instance of $\mathcal{P}_{\texttt{CTL}}$ finalizes pre-certificates of $pid_{\texttt{CA}}$, i. e., $pid_{\texttt{CA}}$ is stored in CAs, and the signature $\sigma_{\texttt{CA}}$ verifies, $\mathcal{P}_{\texttt{CTL}}$ starts the certificate finalization process. Therefore, it firstly queries $\mathcal{F}_{\text{psync-net}}$ for the current time τ. Then, $\mathcal{P}_{\texttt{CTL}}$ adds an own ID (incremental counter), its own identity, and the current timestamp τ to the pre-certificate and then signs $(id_{\texttt{CTL}}, (id, (pid_r, pk_r), pid_{\texttt{CA}}, \sigma_{\texttt{CA}}), pid_{\texttt{CTL}}, \tau)$. Afterwards, it stores the finalized certificate $((id_{\texttt{CTL}}, (id, (pid_r, pk_r), pid_{\texttt{CA}}, \sigma_{\texttt{CA}}), pid_{\texttt{CTL}}), \sigma_{\texttt{CTL}}, \tau)$ in its state.

If a party queries for $\mathcal{P}_{\texttt{CTL}}$'s state via `Read`, $\mathcal{P}_{\texttt{CTL}}$ signs its state and replies to the request with $(\texttt{Read}, \mathsf{pid}_{\texttt{cur}}, \mathsf{state}, \sigma_{\texttt{CTL}})$.

To allow clients to monitor issued certificates, $\mathcal{P}_{\texttt{CTL}}$ provides the `Monitor` interface/command. If $\mathcal{P}_{\texttt{CTL}}$ receives $(\texttt{Monitor}, ctr, pid)$ from a client pid (via $\mathcal{F}_{\text{psync-net}}$), it extracts all certificates issued for pid from its state and sends the certificates (including the used message identifier ctr) back to $\mathcal{F}_{\text{psync-net}}$.

Corruption: CTLs are trusted anchors in our model and thus, $\mathcal{P}_{\texttt{CTL}}$ is incorruptible. Further note that signature keys of CTLs, i. e., their `signer` instances at $\mathcal{F}_{\text{sig}}$, as well as their signature verifying function, i. e., their `verifier` instances at $\mathcal{F}_{\text{sig}}$, cannot be corrupted. This is handled/ensured via $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ and our slightly adapted variant of $\mathcal{F}_{\text{sig}}$.

Remark: Note that one can enhance our model to a setting where only a threshold of CTLs per CA needs to be honest. For the sake of presentation, we decided to present the simpler variant with incorruptible CTLs.

The initialization functionality $\mathcal{F}^{\texttt{CA}}_{\text{init}}$. The initialization functionality $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ (cf. Figure 26) models a trusted setup/distribution of CAs/CTLs identities and their public keys. There is one instance of $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ per $\mathcal{P}^{\text{acc}}_{\text{PKI}}$. In particular, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ includes *(i)* initialization of CAs, resp. $\mathcal{P}^{\texttt{CA}}$, *(ii)* initialization of CTLs, resp. $\mathcal{P}_{\texttt{CTL}}$, and *(iii)* initialization of judges $\mathcal{P}^{\texttt{CA}}_{\text{judge}}$, It further provides a interface for all participants in $\mathcal{P}^{\text{acc}}_{\text{PKI}}$ to query for CA, resp. CTL, data including their public keys and is used to ensure that clients and their accompanied local judges are initialized simultaneously.

If an instance of $\mathcal{P}^{\texttt{CA}}$ asks for initialization, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ queries $\mathcal{A}$ for the CTLs $\mathcal{P}^{\texttt{CA}}$ should use. If one of the provided CTLs is not already registered, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ initializes its session at $\mathcal{F}_{\text{sig}}$. Note that $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ enforces that all used signature keys for CTLs are not corrupted. When storing which CTLs "work" for a dedicated CA, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ checks that every CTLs session at $\mathcal{F}_{\text{sig}}$/signature key is not corrupted. Otherwise, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ declines the initialization of $\mathcal{P}^{\texttt{CA}}$ and queries $\mathcal{A}$ for a new set of CTLs for $\mathcal{P}^{\texttt{CA}}$. If the registration of the CTLs succeeds, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ also initializes $\mathcal{P}^{\texttt{CA}}$ signature key at $\mathcal{F}_{\text{sig}}$ if necessary. All data is stored in $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ state: each CA's pid, resp. CTL, is added to $\mathsf{pid}_{\texttt{CA}}$, resp. $\mathsf{pid}_{\texttt{CTL}}$. The map $\mathsf{pk}_{\texttt{CA}}$, resp. $\mathsf{pk}_{\texttt{CTL}}$, stores the public key of each registered CA, resp. CTL. The map CTLs, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ stores for each CA which CTLs are assigned to it. Finally, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ sends *(i)* the CA's public key, *(ii)* the CTLs the CA uses, and *(iii)* the CTLs' public keys back to the instance of $\mathcal{P}^{\texttt{CA}}$ which asked for initialization.

Similarly, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ initializes an instance of $\mathcal{P}_{\texttt{CTL}}$. This mainly includes to provide the CTL *(i)* its public key, *(ii)* the CAs for which it finalizes pre-certificates, and *(iii)* the relevant CAs public keys (to the dedicated instance of $\mathcal{P}_{\texttt{CTL}}$). The handling is similar to the initialization of a CA. If the CTL/CA is not registered at $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ so far, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ creates a signature key for it at $\mathcal{F}_{\text{sig}}$. Note that $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ ensures that signature keys of CTLs are not corrupted – if a newly registered CTL's $\mathcal{F}_{\text{sig}}$ session is corrupted, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ aborts the registration for this CTL.

When a judge ($\mathcal{P}^{\texttt{CA}}_{\text{judge}}$) queries $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ for initialization, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ simply forwards all its information regarding the PKI session to the judge. This includes, *(i)* the identities (PIDs) of the registered CAs, resp. CTLs, *(ii)* their public keys, and *(iii)* the information which CA and CTLs work together.

Besides initializing parties, parties can also query $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ with *(i)* `GetCaCtls`, *(ii)* `GetCAs`, and *(iii)* `GetCTLs`. If a party queries for $(\texttt{GetCaCtls}, pid_{\texttt{CA}})$, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ replies with the public key of the CA $pid_{\texttt{CA}}$, the (PIDs of the) CTLs of the CA, and the CTLs' public keys. In Case *(ii)*, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ provides all registered (PIDs of) CAs and their public keys to the requestor. In *(iii)*, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ provides all registered (PIDs of) CTLs and their public keys to the requestor.

Further, we use the initialization functionality $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ to ensures that clients and their judges are initialized "at the same time". This simplifies the proof later on, as we do not have to take care of several edge cases in cases of non-initialized parties.[15]

[15]Note that an ITM in iUC which is initialized via a corruption status request always responds that the ITM is not corrupted so far. We want to avoid this edge case in our modeling.

The certificate monitoring enforcing $\mathcal{F}_{\text{psync-net}}$. The network functionality $\mathcal{F}_{\text{psync-net}}$ (cf. Figure 27) ensures that honest parties can check in "near-time" whether there are malicious certificates issued for their identities. Especially for this purpose, $\mathcal{F}_{\text{psync-net}}$ models partially synchronous network between clients and CTLs. There is one instance of $\mathcal{F}_{\text{psync-net}}$ per PKI session.

On every activation, $\mathcal{F}_{\text{psync-net}}$ updates the list of available CTLs at $\mathcal{F}_{\text{init}}^{\mathtt{CA}}$ (via `GetCTLs`) and checks whether all registered honest clients (stored in $\mathsf{pid}_{\text{client}}^{h}$) are still honest (via querying all clients in $\mathsf{pid}_{\text{client}}^{h}$ for their corruption status). If previously honest clients have been corrupted, $\mathcal{F}_{\text{psync-net}}$ removes them from $\mathsf{pid}_{\text{client}}^{h}$ (and also from monitoring in monReg and $\mathsf{requestQueue}$, see below).

When honest parties start a monitoring process, i.e., they query every registered CTL for the certificates published for their identity, they send a `Monitor` message to $\mathcal{F}_{\text{psync-net}}$ including their identity/pid and (the PIDs of) the CTLs which they want to monitor. $\mathcal{F}_{\text{psync-net}}$ stores this request in $\mathsf{requestQueue}$. More specifically, it stores one entry per CTL in $\mathsf{requestQueue}$ including *(i)* an unambiguous identifier for each entry and *(ii)* the time τ (according to $\mathcal{F}_{\text{psync-net}}$'s internal clock) when $\mathcal{F}_{\text{psync-net}}$ received the monitoring request. Additionally, $\mathcal{F}_{\text{psync-net}}$ stores in monReg that $\mathcal{P}_{\text{client}}^{\mathtt{CA}}$ started a monitoring process at time τ. $\mathcal{F}_{\text{psync-net}}$ then leaks the complete $\mathsf{requestQueue}$ to $\mathcal{A}$.

Similar to the initialization of the monitoring process at $\mathcal{P}_{\text{client}}^{\mathtt{CA}}$, $\mathcal{A}$ is expected to trigger further processing of the monitoring at $\mathcal{F}_{\text{psync-net}}$, i.e., *(i)* forwarding a monitoring request to a CTL and *(ii)* deliver the response of a CTL back to the monitoring party. If the adversary triggers $\mathcal{F}_{\text{psync-net}}$ via $(\mathtt{Deliver}, ctr)$, $\mathcal{F}_{\text{psync-net}}$ firstly checks whether a request for the ctr exists in $\mathsf{requestQueue}$ and which of the two cases above needs to be handled. If the request was not handled by the dedicated CTL so far, $\mathcal{F}_{\text{psync-net}}$ adds in $\mathsf{requestQueue}$ a (current) timestamp to entry ctr and forwards the monitor request (including the identifier ctr) to the CTL. In Case *(ii)*, the `Deliver` message of $\mathcal{A}$ triggers the delivery of the (stored) response of the CTL (see below) to the monitoring party. Before $\mathcal{F}_{\text{psync-net}}$ delivers the response, it updates the entry with ID ctr in $\mathsf{requestQueue}$ and adds a (current) timestamp as delivery time.

When a CTL (immediately) responds to the request from Case *(i)* above, it sends $(\mathtt{Monitor}, ctr, resp)$ to $\mathcal{F}_{\text{psync-net}}$. $\mathcal{F}_{\text{psync-net}}$ records the response in the dedicated entry for ctr in $\mathsf{requestQueue}$. Note that *(i)* CTLs are modeled to be incorruptible and *(ii)* CTLs and $\mathcal{F}_{\text{psync-net}}$ have a direct I/O connection for handling monitoring request, i.e., responses from CTLs to $\mathcal{F}_{\text{psync-net}}$ will be delivered "immediately" and cannot be manipulated by $\mathcal{A}$. Thus, $\mathcal{A}$ is not activated when a CTL generates a response to a monitoring request and sends it to $\mathcal{F}_{\text{psync-net}}$. Therefore, we do not store an additional timestamp to determine when the CTL replies to a response. This timestamp is always equal to the timestamp stored in Case *(i)* from above.

The adversary is allowed to update the internal clock of $\mathcal{F}_{\text{psync-net}}$ via the `UpdateRound` command. $\mathcal{F}_{\text{psync-net}}$ accepts a time update (and increases the time by one) iff *(i)* there is no monitoring process initially triggered more than δ time units ago but without response from a CTL, *(ii)* there is no response from a monitoring process waiting for delivery for more then δ time units (both properties can be derived from $\mathsf{requestQueue}$), and *(iii)* every honest party started a monitoring process at least δ time units ago (stored in monReg). $\mathcal{F}_{\text{psync-net}}$ also allows parties to read from its in-build clock (via `GetCurRound`).

The network functionality $\mathcal{F}_{\text{psync-net}}$ is incorruptible.

The local judge $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$. The (local) judge $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$ (cf. Figure 28) handles when a client finds a maliciously issued certificate for its identity.

On every activation, $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$ updates/pulls the list of CAs, resp. CTLs, which are currently available in the PKI, including their public keys (via $\mathtt{InitMe}_{\mathtt{judge}}$ from $\mathcal{F}_{\text{init}}^{\mathtt{CA}}$).

In case that a client finds a certificate during monitoring issued for her identify which she did not request, the client forwards the certificate to her local judge. $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$ verifies that the certificate is valid, i.e., CA and CTL signature verify. As this indicates that the issuing CA misbehaved (at least from the current clients perspective, as clients requests certificates via an authenticated channel at CA), $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$ adds $\mathsf{dis}(pid_{\mathtt{CA}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\mathtt{CA}} : \mathtt{ca})$ (where $(pid_{\mathtt{CA}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\mathtt{CA}} : \mathtt{ca})$ denotes the CA that attested the reported maliciously generated certificate) to her (local) verdict. Note that external observers or the environment cannot trust the verdict of $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$. From a higher-level perspective, one cannot be sure which of the two parties, the CA or the client, indeed misbehaved in this case. A malicious client (in turn a malicious judge) could have requested the certificate at an honest CA but then blames the CA that she did not request the certificate. Vice versa, CAs can simply generate certificates for any client. $\mathcal{P}_{\text{judge}}^{\mathtt{CA}}$ also marks `correctCert` broken for its associated client.

$\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ implements the `GetVerdict` interface which simply outputs the stored local verdict and a interface to access a judicial report. On receiving `GetJudicialReport`, $\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ does not provide information and outputs ε.

Corrupting a local judge as long its accompanied client party is still honest is not allowed. This models that a clients validation routine still works honestly/correctly as long as the accompanied client is honest/works correctly. Note that $\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ is a pure message forwarder to/from $\mathcal{A}$ in case that the associated client is corrupted.

The public supervisor $\mathcal{P}^{\mathtt{CA}}_{\mathrm{sv}}$. As we do not model assumption-based security properties in this case study, the purpose of the supervisor $\mathcal{P}^{\mathtt{CA}}_{\mathrm{sv}}$ (cf. Figure 29) is to provide the environment access to the corruption status of internal protocol participants (here: CAs). Thus, on request $(\mathtt{corruptInt}?, (pid, sid, role))$ where $role$ needs to be $\mathtt{ca}$, $\mathcal{P}^{\mathtt{CA}}_{\mathrm{sv}}$ queries $(pid, sid, role)$ for its corruption status and forwards the result to the requestor.

Note that we include a "dummy" interface for `IsAssumptionBroken`?. On every request, $\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ simply replies "`false`".

The ideal authenticated channel functionality $\mathcal{F}_{\mathrm{auth}}$. The authenticated channel functionality $\mathcal{F}_{\mathrm{auth}}$ (cf. Figure 30) is derived from Canetti's message authentication functionality [22] in combination with the secure channel functionality $\mathcal{F}_{\text{sec-channel}}$ from [70]. In our case study, we use $\mathcal{F}_{\mathrm{auth}}$ to authenticate clients to CAs. Thus, CAs can be sure that a client indeed registers a certificate for its identity.

One instance of $\mathcal{F}_{\mathrm{auth}}$ is meant to handle all authenticated channels in an instance/session of $\mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}}$. The whole session of $\mathcal{F}_{\mathrm{auth}}$ can be corrupted (as long as there are no channels initialized yet). Parties "register" at $\mathcal{F}_{\mathrm{auth}}$ via `Establish`. The adversary may trigger the finalization the party's registration. Parties can send messages to specified receivers via $\mathcal{F}_{\mathrm{auth}}$. $\mathcal{F}_{\mathrm{auth}}$ buffers the messages and $\mathcal{A}$ may trigger the delivery of the message. Note that $\mathcal{A}$ has full control over the messages send to/delivered from $\mathcal{F}_{\mathrm{auth}}$ when $\mathcal{F}_{\mathrm{auth}}$ is corrupted. Furthermore, $\mathcal{A}$ may drop messages which are queued for delivery in $\mathcal{F}_{\mathrm{auth}}$.

The ideal signature functionality $\mathcal{F}_{\mathrm{sig}}$. The ideal signature functionality $\mathcal{F}_{\mathrm{sig}}$ (cf. Figure 31) is mainly the same as presented in [17] and matches common approaches for ideal signature functionalities.

As we need to ensure in $\mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}}$ that CTL signatures are incorruptible, $\mathcal{F}_{\mathrm{sig}}$ is additionally connected to $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$ to determine whether a party is a CTL. $\mathcal{F}_{\mathrm{sig}}$ declines corruption of CTLs.

In contrast to [17], we do not allow $\mathcal{A}$ to corrupt `verifier` instances of $\mathcal{F}_{\mathrm{sig}}$ as one typically expects that signature verification is a local process and not externalized and thus honest as long the considered party is honest. Indeed, we need to ensure for the model that at least `verifier` instance of (local) judges need to be honest as long as the accompanied judge is honest. Otherwise, the security proof would fail.

E.3 UC Security Analysis: We now present the formal variant of Theorem 2 and its security proof in detail.

Theorem 7. *Let $\mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}}$ be as defined above and let $\delta \in \mathbb{N}$ be the upper time boundary for message delivery in $\mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}}$, resp. $\mathcal{F}_{\mathrm{psync\text{-}net}}$, and $\Sigma = (\mathsf{gen}(1^\eta), \mathsf{sig}, \mathsf{ver})$ be an EUF-CMA secure signature scheme.*

Let $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ be as described above with $\mathcal{F}^{\mathrm{acc\text{-}PKI}}_{\mathrm{judgeParams}}$ as subroutine with parameters $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{correctCert}\}$, $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$, $\mathsf{pids}_{\mathsf{judge}} = \{\mathtt{local}\} \times \{0,1\}^* \times \{\mathtt{client}\}$, and $\mathsf{ids}_{\mathsf{assumption}} = \emptyset$. Then:

$$(\mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}}) \leq (\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}} \mid \mathcal{F}^{\mathrm{acc\text{-}PKI}}_{\mathrm{judgeParams}})$$

We remark that the following proof repeats many general points/techniques from the proof of Theorem 6, e.g., regarding the simulator, in verbatim. This is mainly due to the fact that both protocols – the scaling protocol and the PKI protocol – rely on some kind of global state. In the scaling protocol, the global state is a sequence of totally ordered transactions. In the PKI protocol, the global state consists of a set of certificates.

Proof.
Firstly, we define a responsive simulator $\mathcal{S}$ such that the real world running the protocol $\mathcal{R} := \mathcal{P}^{\mathrm{acc}}_{\mathrm{PKI}} = (\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}} : \mathtt{client}, \mathcal{P}^{\mathtt{CA}}_{\mathrm{sv}} : \mathtt{supervisor}, \mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}} : \mathtt{judge} \mid \mathcal{P}^{\mathtt{CA}} : \mathtt{ca}, \mathcal{P}_{\mathtt{CTL}} : \mathtt{ctl}, \mathcal{F}_{\mathrm{sig}}, \mathcal{F}_{\mathrm{psync\text{-}net}} : \mathtt{psync\text{-}net}, \mathcal{F}_{\mathrm{auth}} : \mathtt{auth})$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$, with the protocol $\mathcal{I} := (\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}} \mid \mathcal{F}^{\mathrm{acc\text{-}PKI}}_{\mathrm{judgeParams}})$, for every ppt environment $\mathcal{E}$.

The simulator $\mathcal{S}$ is defined as follows: as common, $\mathcal{S}$ is a single machine. It is connected to $\mathcal{I}$ and the environment $\mathcal{E}$ via their network interfaces. In a run, there is only a single instance of the machine $\mathcal{S}$ that accepts and processes all incoming messages. The simulator $\mathcal{S}$ internally simulates the realization $\mathcal{R}$, including its behavior on the network interface connected to the environment, and uses this simulation to compute responses

Description of a CA client $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}} = (\mathtt{client})$:

```
Participating roles: {client}
Corruption model: Dynamic corruption without secure erasures
Protocol parameters:
    – $\delta \in \mathbb{N}$                                                    {Network delay
```

Description of $M_{\mathtt{client}}$:

```
Implemented role(s): {client}
Subroutines: $\mathcal{F}_{\mathrm{auth}}$ : auth, $\mathcal{F}_{\mathrm{sig}}$ : signer, $\mathcal{F}_{\mathrm{sig}}$ : verifier, $\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ : judge, $\mathcal{F}_{\mathrm{psync\text{-}net}}$ : psync-net, $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$ : init
Internal state:
    – $\mathsf{pk} \in \{0,1\}^*, \mathsf{pk} = \bot$                                    {Public key of the client
    – $\mathsf{certCA} \in \{0,1\}^*, \mathsf{certCA} = \bot$                            {The CA which should create $\mathsf{pid}_{\mathsf{cur}}$'s certificate
    – $\mathsf{requests} \subset \{0,1\}^* \times (\{0,1\}^*)^3, \mathsf{requests} = \emptyset$   {The recorded requests of $\mathsf{pid}_{\mathsf{cur}}$ including requesting party.
    – $\mathsf{pk}_{\mathtt{CA}} : \{0,1\}^* \to \{0,1\}^*$                              {The pubkeys of the CAs, initially $\bot$
    – $\mathsf{CTLs} : \{0,1\}^* \to \{0,1\}^*$                                      {The CTLs of each CA, initially $\bot$
    – $\mathsf{pk}_{\mathtt{CTL}} : \{0,1\}^* \to \{0,1\}^*$                             {Pubkeys of CTLs, initially $\bot$
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Corruption behavior:
    DetermineCorrStatus($pid, sid, role$):
        if corrupted = true:                                               {Checks whether party itself is corrupted.
            return true
        $corrRes_1 \leftarrow$ corr($\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}})$, signer)          {Request corruption status at $\mathcal{F}_{\mathrm{sig}}$
        send CorruptionStatus? to $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}}$ : auth)          {Query $\mathcal{F}_{\mathrm{auth}}$ for its corruption status.
        wait for (CorruptionStatus?, $corrRes_2$)
        return $(corrRes_1 \lor corrRes_2)$                         {Return whether $\mathcal{F}_{\mathrm{sig}}$ or $\mathcal{F}_{\mathrm{auth}}$ instances are corrupted
EntityInitialization:
    send Establish to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{psync\text{-}net}}$ : psync-net); wait for _     {Register party at $\mathcal{F}_{\mathrm{psync\text{-}net}}$
    if ITM was not activated via init macro, resp. a InitEntity message:
        send initPartner to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$ : init)
        wait for initPartner
Main:
    recv (Register, $pk, pid_{\mathtt{CA}}$) from I/O:                              {Client should register certificate
        if $\mathsf{pk} = \varepsilon$:                                                   {Store public key.
            $\mathsf{pk} \leftarrow pk, \mathsf{certCA} \leftarrow pid_{\mathtt{CA}}$
            send (Send, $(pid_{\mathtt{CA}}, \mathsf{sid}_{\mathsf{cur}}$, ca), (Register, $\mathsf{pid}_{\mathsf{cur}}, \mathsf{pk}$)) to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}}$ : auth)
    recv (Retrieve, $pid, pid_{\mathtt{CA}}$) from I/O:                             {Query for certificate
        requests.add($[pid, pid_{\mathtt{CA}}], (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$)
        send (Retrieve, $pid, pid_{\mathtt{CA}}$) to NET        {Retrieve requests are dispatched via $\mathcal{A}$, i.e., there are no delivery gura
    recv (Retrieve, $(ctr, (serial, pid, pk, pid_{\mathtt{CA}}, \sigma_{\mathtt{CA}}), pid_{\mathtt{CTL}}, \tau, \sigma_{\mathtt{CTL}})$) from NET:
        send (GetCaCtls, $pid_{\mathtt{CA}}$) to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$ : init)       {Query trusted CA/CTL information at $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$
        wait for (GetCaCtls, $pk_{\mathtt{CA}}, (pid_1, \ldots, pid_n), (pk_1, \ldots, pk_n)$)
        $\mathsf{pk}[pid_{\mathtt{CA}}] \leftarrow pk_{\mathtt{CA}}$                                              {Store trusted CA public key
        $\mathsf{CTLs}[pid_{\mathtt{CA}}] \leftarrow (pid_1, \ldots, pid_n)$
        for $i \in \{1, \ldots, n\}$ do:                                            {Store CTL information
            $\mathsf{pk}_{\mathtt{CTL}}[pid_i] \leftarrow pk_i$
        send GetCurRound to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}$, psync-net)                          {Get current time
        wait for (GetCurRound, $\tau'$)
        if $\exists([pid, pid_{\mathtt{CA}}], (pid_c, sid_c, role_c)) \in \mathsf{requests} \land pid_{\mathtt{CTL}}$ is in $\mathsf{CTLs}[pid_{\mathtt{CA}}] \land \tau' > 3 \cdot \delta + \tau$:
            {Handle answer only if there is a matching request and $pid_{\mathtt{CTL}}$ is registered as CTL for $pid_{\mathtt{CA}}$ and certificate is sufficient old
             to be considered valid
            $b_1 \leftarrow$ verifySig($[ctr, (serial, pid, pk, pid_{\mathtt{CA}}, \sigma_{\mathtt{CA}}), pid_{\mathtt{CTL}}, \tau], \sigma_{\mathtt{CTL}}, \mathsf{pk}_{\mathtt{CTL}}[pid_{\mathtt{CTL}}], pid_{\mathtt{CTL}}$)   {Check CTL signature
            $b_2 \leftarrow$ verifySig($[serial, pid, pk, pid_{\mathtt{CA}}], \sigma_{\mathtt{CA}}, \mathsf{pk}_{\mathtt{CA}}[pid_{\mathtt{CA}}], pid_{\mathtt{CA}}$)                  {Check CA signature
            if $b_1 \land b_2$:
                requests.remove[$([pid, pid_{\mathtt{CA}}], [pid_c, sid_c, role_c])$]                     {Remove request
                send (Retrieve, $pid, pid_{\mathtt{CA}}, pk$) to $(pid_c, sid_c, role_c)$
    recv Establish from NET:                                        {$\mathcal{A}$ triggers the registration at $\mathcal{F}_{\mathrm{auth}}$
        send (Establish, $\varepsilon$) to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}}$ : auth)
```

Fig. 22: The CA client $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ (Part 1).

Description of $M_{\mathtt{client}}$ (continued):

```
Main:
    recv Monitor from NET:                                                    {$\mathcal{A}$ triggers certificate monitoring
        send GetCTLs to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}^{\mathtt{CA}}_{\mathrm{init}} : \mathtt{init})$    {Request CTLs at $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$
        wait for $(\mathtt{GetCTLs}, CTL, pid_{\mathtt{CTL}})$
        send $(\mathtt{Monitor}, \mathsf{pid_{cur}}, CTL)$ to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\mathrm{psync\text{-}net}} : \mathtt{psync\text{-}net})$    {Send request to $\mathcal{F}_{\mathrm{psync\text{-}net}}$
    recv $(\mathtt{Monitor}, ctr, msg)$ from NET:                                {Check monitored certificates
        send GetCAs to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}^{\mathtt{CA}}_{\mathrm{init}} : \mathtt{init})$    {Request CAs at $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$
        wait for $(\mathtt{GetCAs}, CA, pk_{\mathtt{CA}})$
        send GetCTLs to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}^{\mathtt{CA}}_{\mathrm{init}} : \mathtt{init})$    {Request CTLs at $\mathcal{F}^{\mathtt{CA}}_{\mathrm{init}}$
        wait for $(\mathtt{GetCTLs}, CTL, pk_{\mathtt{CTL}})$
        if $msg = \{(ctr', [serial, \mathsf{pid_{cur}}, pk, pid'_{\mathtt{CA}}, \sigma_{\mathtt{CA}}], pid'_{\mathtt{CTL}}, \tau, \sigma_{\mathtt{CTL}}), \ldots\}$:    {Verify signatures
            for all $(ctr', [serial, \mathsf{pid_{cur}}, pk, pid'_{\mathtt{CA}}, \sigma_{\mathtt{CA}}], pid_{\mathtt{CTL}},', \tau, \sigma_{\mathtt{CTL}})$ from above set do:
                $b \leftarrow \mathtt{verifySig}([(ctr', [serial, \mathsf{pid_{cur}}, pk, pid'_{\mathtt{CA}}, \sigma_{\mathtt{CA}}], pid_{\mathtt{CTL}}, \tau], \sigma_{\mathtt{CTL}}, pk_{\mathtt{CTL}}[pid'_{\mathtt{CTL}}], pid'_{\mathtt{CTL}})$    {Check CTL signature
                if $pid'_{\mathtt{CA}} \in CA$:
                    $c \leftarrow \mathtt{verifySig}([serial, \mathsf{pid_{cur}}, pk, pid'_{\mathtt{CA}}], \sigma_{\mathtt{CA}}, pk_{\mathtt{CA}}[pid'_{\mathtt{CA}}], pid'_{\mathtt{CA}})$    {Check CA signature
                else:
                    $c \leftarrow \mathtt{false}$
                if $b \wedge c$:                                                  {Certificate is validly signed
                    if $pk \neq \mathsf{pk}$:
                        send $(\mathtt{Evidence}_M, (ctr', [serial, pid, pk, pid'_{\mathtt{CA}}, \sigma_{\mathtt{CA}}], pid'_{\mathtt{CTL}}, \tau, \sigma_{\mathtt{CTL}}))$ to $((\mathtt{local}, pid, \mathtt{client}), \mathsf{sid_{cur}}, \mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$ :
    judge)
                                                                              {Forward evidence to $\mathcal{P}^{\mathtt{CA}}_{\mathrm{judge}}$
                        // In the model: stop processing after finding the first maliciously generated certificate //
Procedures and Functions:
    function verifySig$(msg, \sigma, pk, pid)$ :                                {Verify signature at $\mathcal{F}_{\mathrm{sig}}$
        send $(\mathtt{VerResult}, msg, \sigma, pk)$ to $((\mathsf{pid_{cur}}, \mathtt{ctl}), (pid, \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{sig}} : \mathtt{verifier})$; wait for $(\mathtt{VerResult}, result)$
        return $result$
```

Fig. 23: The CA client $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ (Part 2).

to incoming messages. For ease of presentation, we will refer to this internal simulation by $\mathcal{R}'$. More precisely, the simulation runs as follows:

Network communication from/to the environment

- Messages that $\mathcal{S}$ receives on the network interface (connected to the environment) and which are hence meant for $\mathcal{R}$ are forwarded to the internal simulation $\mathcal{R}'$.
- Any messages sent by $\mathcal{R}'$ on its network interface (that are hence meant for the environment) are forwarded to the environment $\mathcal{E}$.

Corruption handling

- The simulator $\mathcal{S}$ keeps the corruption status of entities in $\mathcal{R}'$ and $\mathcal{I}$ synchronized. That is, whenever an entity of $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ or $\mathcal{P}^{\mathtt{CA}}$ in $\mathcal{R}'$ starts to consider itself corrupted, the simulator corrupts the corresponding (internal) entity of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ in $\mathcal{I}$ before continuing its simulation. Note that corruption of internal entities, i. e., of $\mathcal{P}^{\mathtt{CA}}$, is mapped to a corruption of an internal party in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ in $\mathcal{I}$.
- Incoming messages from corrupted (non-internal) entities of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ in $\mathcal{I}$ are forwarded on the network interface to the environment in the name of the corresponding entity/instance of $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ in $\mathcal{R}'$. Conversely, whenever a corrupted entity of $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ wants to output a message to a higher-level protocol, $\mathcal{S}$ instructs the corresponding entity of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ to output the same message to the higher-level protocol.

Certificate registration

Whenever an honest entity $entity = (pid, sid, \mathtt{client})$ of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ receives a request $(\mathtt{Register}, pk, pid_{\mathtt{CA}})$ to register a certificate, resp. a public key, for its own identity pid, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ buffers the request in $\mathsf{buffer_S}$ and leaks the full message buffer to $\mathcal{S}$. The simulator $\mathcal{S}$ uses the leaked message to simulate the registration of pk for pid at $pid_{\mathtt{CA}}$ in $\mathcal{R}'$, i.e, it simulates the input of $(\mathtt{Register}, pk, pid_{\mathtt{CA}})$ to the $(pid, sid, \mathtt{client})$'s instance of $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ in $\mathcal{R}'$.

We remark that $\mathcal{S}$ does not need to handle the registration of certificates from corrupted parties. $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ does not provide guarantees for these certificates and allows $\mathcal{A}$ to freely determine the output in case that a party requests the certificate of a corrupted party.

Retrieve requests

Whenever the environment instructs an honest entity $(pid, sid, \mathtt{client})$ via $(\mathtt{Retrieve}, pid_{req}, pid_{\mathtt{CA}})$ to request a certificate of pid_{req} from $pid_{\mathtt{CA}}$, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ buffers this request (in $\mathsf{buffer_R}$) and leaks the full request to $\mathcal{S}$. Upon receiving the leakage from $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$, $\mathcal{S}$ forwards this message to $\mathcal{P}^{\mathtt{CA}}_{\mathrm{client}}$ in $\mathcal{R}'$ and simulates its behavior.

Description of a CA $\mathcal{P}^{\texttt{CA}} = (\texttt{ca})$:

Participating roles: $\{\texttt{ca}\}$
Corruption model: *Dynamic corruption without secure erasures*

Description of $M_{\texttt{ca}}$:

```
Implemented role(s): $\{\texttt{ca}\}$
Subroutines: $\mathcal{F}_{\text{init}}^{\texttt{CA}}$ : init, $\mathcal{F}_{\text{sig}}$ : signer, $\mathcal{F}_{\text{sig}}$ : verifier, $\mathcal{P}_{\text{judge}}^{\texttt{CA}}$ : judge
Internal state:
    – $\mathsf{pk}_{\mathsf{CA}} \in \{0,1\}^*, \mathsf{pk}_{\mathsf{CA}} = \varepsilon$          {The public key of CA $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$
    – $\mathsf{CTLs} \subset \{0,1\}^*, \mathsf{CTLs} = \emptyset$          {The CA's CTL's pids
    – $\mathsf{pk}_{\mathsf{CTL}} : \mathsf{CTLs} \to \{0,1\}^*$          {CTLs' pubkeys, initially $\bot$
    – $\mathsf{serial} \in \mathbb{N}, \mathsf{serial} = 0$          {The serial number of certificates issued by entity is a counter
    – $\mathsf{certs} \subseteq \mathbb{N} \times (\{0,1\}^*)^4, \mathsf{certs} = \emptyset$          {CAs issued certificates of form $(serial, subject, pk, issuer, \sigma)$
    – $\mathsf{certs}_{\mathsf{CTL}} : \mathsf{CTLs} \to \{0,1\}^*$          {The state of each CTL after reading initially $\bot$
CheckID$(pid, sid, role)$:
    Accept all messages for $(pid, sid, role)$ where $sid = (pid, sid')$ and $role = \texttt{ca}$.
    Do not accept an entity $(pid, sid, role)$ if $pid$ can be parsed to $(pid', \texttt{client})$          {Avoid local judges like PIDs
Corruption behavior:
    DetermineCorrStatus$(pid, sid, role)$:
        if corrupted = true:          {Checks whether party itself is corrupted.
            return true
        $corrRes_1 \leftarrow \mathbf{corr}(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \texttt{signer})$          {Request corruption status at $\mathcal{F}_{\text{sig}}$
        send CorruptionStatus? to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$          {Query $\mathcal{F}_{\text{auth}}$ for its corruption status.
        wait for (CorruptionStatus?, $corrRes_2$)
        $corrRes_3 \leftarrow \texttt{false}$          {Check that all used verifier instances are not corrupted
        for all queries to $(\mathsf{pid}_{\mathsf{cur}}, (pid', sid'), \texttt{verifier})$ at $\mathcal{F}_{\text{sig}}$ from transcript do:
            $corrRes_3 = corrRes_3 \lor \mathbf{corr}(\mathsf{pid}_{\mathsf{cur}}, (pid', sid'), \texttt{verifier})$
        return $(corrRes_1 \lor corrRes_2 \lor corrRes_3)$          {Return whether $\mathcal{F}_{\text{sig}}$ or $\mathcal{F}_{\text{auth}}$ instances are corrupted
Initialization:
        send InitMe$_{\mathsf{CA}}$ to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init}}^{\texttt{CA}} : \texttt{init})$          {Request CA setup at $\mathcal{F}_{\text{init}}^{\texttt{CA}}$
        wait for $(\texttt{InitMe}_{\mathsf{CA}}, pk_{\mathsf{CA}}, (pid_1, \dots, pid_n), (pk_1, \dots, pk_n), pk_{ctl})$
        $\mathsf{pk}_{\mathsf{CA}} \leftarrow pk_{\mathsf{CA}}$          {Store (trusted) public keys and used CTLs
        $\mathsf{CTLs} \leftarrow \{pid_1, \dots, pid_n\}$
        for all $i \in \{1, \dots, n\}$ do:
            $\mathsf{pk}_{\mathsf{CTL}}[pid_i] \leftarrow pk_i$
Main:
        recv (Received, $(pid, \mathsf{sid}_{\mathsf{cur}}, role)$, (Register, $pid, pk$)) from I/O:          {$pid$ registers public key/certificate via $\mathcal{F}_{\text{auth}}$
            if $b \land \nexists(\_, pid, \_, \_, \_) \in \mathsf{certs}$:          {One certificate per party
                $\mathsf{serial} \leftarrow \mathsf{serial} + 1$
                $cert' \leftarrow (\mathsf{serial}, pid, pk, \mathsf{pid}_{\mathsf{cur}})$          {Simplified version of a certificate for pid w/o signature
                $\sigma' \leftarrow \texttt{sign}(cert')$          {Generate signature for certificate
                $\mathsf{certs}.\mathsf{add}([cert', \sigma'])$          {Record certificate
                send (Submit, $cert', \sigma'$, CTLs) to NET          {Publish certificate at CTLs
        recv (Read, $pid_{\mathsf{CTL}}, state, \sigma$) from NET s.t. $pid_{\mathsf{CTL}} \in \mathsf{CTLs}$:          {Current state of CTL pid
            $b \leftarrow \texttt{verifySig}(state, \sigma, \mathsf{pk}_{\mathsf{CTL}}[pid_{\mathsf{CTL}}], pid_{\mathsf{CTL}})$          {Verify signature
            if $b \land \mathsf{certs}_{\mathsf{CTL}}[pid] \subset state$:          {$\mathcal{P}^{\texttt{CA}}$ only accepts valid states which extend the current state
                $check \leftarrow \texttt{true}$
                for all $(ctr, cert, pid, \sigma) \in state$ do:
                    if $pid \neq pid_{\mathsf{CTL}}$:          {A CTL only has its "own" certificates
                        break
                    $b \leftarrow \texttt{verifySig}([ctr, cert], \sigma, \mathsf{pk}_{\mathsf{CTL}}[pid], pid)$
                    if $\neg b$:
                        $check \leftarrow check \lor b$
                if $check$:          {CTL list is valid
                    $\mathsf{certs}_{\mathsf{CTL}}[pid] \leftarrow state$
        recv (Retrieve, $pid, \mathsf{pid}_{\mathsf{cur}}$) from NET:          {Client queries for certificate
            if $\exists(serial, pid, pk, \mathsf{pid}_{\mathsf{cur}}, \sigma) \in \mathsf{certs}$:
                if $\exists$ $\mathsf{certs}_{\mathsf{CTL}}[pid_{\mathsf{CTL}}]$ of a CTL $pid_{\mathsf{CTL}}$, s.t. $(ctr, (serial, pid, pk, \mathsf{pid}_{\mathsf{cur}}, \sigma), pid_{\mathsf{CTL}}, \tau, \sigma_{\mathsf{CTL}}) \in \mathsf{certs}_{\mathsf{CTL}}[pid_{\mathsf{CTL}}]$:
                                                                                    {Let $pid_{\mathsf{CTL}}$ be the first hit during search
                    reply (Retrieve, $((ctr, (serial, pid, pk, \mathsf{pid}_{\mathsf{cur}}, \sigma), pid_{\mathsf{CTL}}, \tau, \sigma_{\mathsf{CTL}}))$)
        recv Establish from NET:          {$\mathcal{A}$ triggers the registration at $\mathcal{F}_{\text{auth}}$
            send (Establish, $\varepsilon$) to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$
Procedures and Functions:
        function verifySig$(msg, \sigma, pk, pid)$ :          {Verify signature at $\mathcal{F}_{\text{sig}}$
            send (VerResult, $msg, \sigma, pk$) to $((\mathsf{pid}_{\mathsf{cur}}, \texttt{ctl}), (pid, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{verifier})$
            wait for (VerResult, $result$)
            return $result$
        function sign$(msg)$ :          {Sign message at $\mathcal{F}_{\text{sig}}$
            send (sign, $msg$) to $((\mathsf{pid}_{\mathsf{cur}}, \texttt{ctl}), (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$
            wait for (sign, $\sigma$)
            return $\sigma$
```

Fig. 24: The model of a CA $\mathcal{P}^{\texttt{CA}}$.

Description of the CTL $\mathcal{P}_{\text{CTL}} = (\texttt{ctl})$:

Participating roles: $\{\texttt{ctl}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{ctl}}$:

```
Implemented role(s): $\{\texttt{ctl}\}$
Subroutines: $\mathcal{F}_{\text{sig}} : \texttt{verifier}, \mathcal{F}_{\text{sig}} : \texttt{signer}, \mathcal{F}^{\texttt{CA}}_{\text{init}} : \texttt{init}$
Internal state:
    – $\mathsf{state} \subset \mathbb{N} \times \{0,1\}^* \times \{\mathsf{pid_{cur}}\} \times \mathbb{N} \times \{0,1\}^*, \mathsf{state} = \emptyset$    {The set of certificates of form $(ctr, cert, \mathsf{pid_{cur}}, \tau, \sigma_{\text{CTL}})$
    – $\mathsf{CAs} \subset \{0,1\}^*, \mathsf{CAs} = \emptyset$    {The set of CAs, the CTL will finalize pre-certificates for
    – $\mathsf{pk_{CTL}} \in \{0,1\}^*$    {The public key of the CTL
    – $\mathsf{pk_{CA}} : \mathsf{CAs} \to \{0,1\}^*$    {The (trusted) public keys of the CA
    – $\mathsf{counter} \in \mathbb{N}, \mathsf{counter} = 0$    {Serial
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Initialization:
        send $\texttt{InitMe}_{\text{CTL}}$ to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}^{\texttt{CA}}_{\text{init}} : \texttt{init})$    {Request initialization at $\mathcal{F}^{\texttt{CA}}_{\text{init}}$
        wait for $(\texttt{InitMe}_{\text{CTL}}, pk_{\text{CTL}}, (pid_1, \ldots, pid_n), (pk_1, \ldots, pk_n))$
        $\mathsf{pk_{CTL}} \leftarrow pk_{\text{CTL}}$
        $\mathsf{CAs} \leftarrow \{pid_1, \ldots, pid_n\}$    {Store (trusted) CAs and their public keys
        for all $i \in \{1 \ldots, n\}$ do:
            $\mathsf{pk_{CA}}[pid_i] \leftarrow pk_i$
Main:
        recv $(\texttt{Submit}, cert)$ from NET:    {Certificate submission
            require: $cert = (serial, pid_r, pk_r, pid_{\texttt{CA}}, \sigma_{\texttt{CA}})$, s.t. $pid_{\texttt{CA}} \in \mathsf{CAs}$
            $b \leftarrow \texttt{verifySig}((serial, pid_r, pk_r, pid_{\texttt{CA}},), \sigma_{\texttt{CA}}, \mathsf{pk}[pid_{\texttt{CA}}], pid_{\texttt{CA}})$    {Check valid signature
            if $b$:    {If signature is valid, record data in state
                send GetCurRound to $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\text{psync-net}} : \texttt{psync-net})$    {Query current time at $\mathcal{F}_{\text{psync-net}}$
                wait for $(\texttt{GetCurRound}, \tau)$
                $\mathsf{counter} \leftarrow \mathsf{counter} + 1$
                $\sigma_{\text{CTL}} \leftarrow \texttt{sign}([\mathsf{counter}, cert, \mathsf{pid_{cur}}, \tau])$    {Mark certificate as valid
                $\mathsf{state.add}([\mathsf{counter}, cert, \mathsf{pid_{cur}}, \tau, \sigma_{\text{CTL}}])$
        recv Read:    {Read request
            $\sigma \leftarrow \texttt{sign}(\mathsf{state})$    {Sign full state
            reply $(\texttt{Read}, \mathsf{pid_{cur}}, \mathsf{state}, \sigma)$    {Reply with full state including signatures
        recv $(\texttt{Monitor}, pid, ctr)$ from I/O:    {Monitor maliciously published certificates
            $resp \leftarrow \varepsilon$
            for all $(ctr', [serial, pid, pk, pid_{\texttt{CA}}, \sigma_{\texttt{CA}}], \mathsf{pid_{cur}}, \tau, \sigma_{\text{CTL}})$ in state do:    {Collect certificates for $pid$
                $resp.\mathsf{add}(ctr', [serial, pid, pk, pid_{\texttt{CA}}, \sigma_{\texttt{CA}}], \mathsf{pid_{cur}}, \tau, \sigma_{\text{CTL}})$
            reply $(\texttt{Monitor}, ctr, resp)$    {Return certificate for $pid$ to requestor
Procedures and Functions:
        function $\texttt{verifySig}(msg, \sigma, pk, pid)$ :    {Verify signature at $\mathcal{F}_{\text{sig}}$
            send $(\texttt{VerResult}, msg, \sigma, pk)$ to $((\mathsf{pid_{cur}}, \mathsf{role_{cur}}), (pid, \mathsf{sid_{cur}}), \mathcal{F}_{\text{sig}} : \texttt{verifier})$
            wait for $(\texttt{VerResult}, result)$
            return $result$
        function $\texttt{sign}(msg)$ :    {Sign message at $\mathcal{F}_{\text{sig}}$
            send $(\texttt{sign}, msg)$ to $((\mathsf{pid_{cur}}, \mathsf{role_{cur}}), (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$
            wait for $(\texttt{sign}, \sigma)$
            return $\sigma$
```

Fig. 25: The simplified CTL $\mathcal{P}_{\text{CTL}}$.

If $\mathcal{A}$ triggers a (simulated) entity $(pid, sid, \texttt{client})$ to respond on a retrieval request by sending $(\texttt{Retrieve}, cert)$ to the entity, $\mathcal{S}$ simulates the input the message in $\mathcal{R}'$. If $(pid, sid, \texttt{client})$ (in $\mathcal{R}'$) wants to output $cert$, resp. parts of $cert$ within a message of form $(\texttt{Retrieve}, pid, pid_{\texttt{CA}}, pk)$ to $(pid_c, sid_c, role_c)$ at the end of its activation, $\mathcal{S}$ sends $(\texttt{Deliver}, pid, pid_{\texttt{CA}}, pid_c, role_c, pk)$ to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. This triggers $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ to deliver the retrieval response. Note, that in case of $\mathsf{brokenProps}[(\texttt{correctCert}, (pid, \mathsf{sid_{cur}}, \texttt{client}))] = \texttt{true}$, the output is forwarded without further checks. If $\texttt{correctCert}$ still holds true, $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ will only forward the output, if a matching request in $\mathsf{buffer_R}$ exists.

State updates
When a CTL finalizes an *uncorrupted* party's certificate (for which $\texttt{correctCert}$ still holds true) in $\mathcal{R}'$, $\mathcal{S}$ triggers an update at $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ to include the finalized certificate in $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. Therefore, it firstly extracts the newly generated certificate from the CTL's state. In particular, $\mathcal{S}$ extracts the identity pid for which the certificate is issued, the used public key pk as well as the used CA $pid_{\texttt{CA}}$. $\mathcal{S}$ then sends $(\texttt{Update}, \{(pid, pk, pid_{\texttt{CA}})\})$ to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$.

Description of the initialization machine $\mathcal{F}_{\text{init}}^{\texttt{CA}} = (\texttt{init})$:

```
Participating roles: $\{\texttt{init}\}$
Corruption model: *incorruptible*
```

Description of $M_{\texttt{init}}$:

```
Implemented role(s): $\{\texttt{init}\}$
Subroutines: $\mathcal{F}_{\text{sig}} : \texttt{signer}$
Internal state:
    – $\mathsf{pid}_{\texttt{CA}} \subset \{0,1\}^*, \mathsf{pid}_{\texttt{CA}} = \emptyset$                    {The pids of the CAs
    – $\mathsf{pk}_{\texttt{CA}} : \mathsf{pid}_{\texttt{CA}} \to \{0,1\}^*$                    {The public keys of CAs, initially $\perp$ for all entries
    – $\mathsf{pid}_{\texttt{CTL}} \subset \{0,1\}^*, \mathsf{pid}_{\texttt{CTL}} = \emptyset$                    {The pids of the CTLs
    – $\mathsf{pk}_{\texttt{CTL}} : \mathsf{pid}_{\texttt{CTL}} \to \{0,1\}^*$                    {The public keys of CTLs, initially $\perp$ for all entries
    – $\mathsf{CTLs} : \mathsf{pid}_{\texttt{CA}} \to (\mathsf{pid}_{\texttt{CTL}} \cup \{\perp\})^*$                    {A CAs trusted CTLs, intitally $\perp$
    – $\mathsf{caller} \in (\{0,1\}^*)^3 \cup \{\perp\} = \perp$                    {Stores the callers during client/judge initialization.
CheckID$(pid, sid, role)$:
    Accept all messages for the same $sid$.
Main:
    recv $\texttt{InitMe}_{\texttt{CA}}$ from $\texttt{I/O}$:                    {CA request initialization
        send responsively $(\texttt{InitMe}_{\texttt{CA}})$ to $\texttt{NET}$ $(\star)$                    {Query $\mathcal{A}$ for initialization details of $\mathsf{pid}_{\mathsf{cur}}$
        wait for $(\texttt{InitMe}_{\texttt{CA}}, pid_1, \ldots, pid_n)$
        if $n < 1$:
            go to $(\star)$
        for all $i \in \{1, \ldots, n\}$ do:
            $corrRes \leftarrow \mathbf{corr}(pid_i, (pid_i, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$                    {Check whether CTL key is still uncorrupted
            if $corrRes$:
                go to $(\star)$
            if $\mathsf{pk}_{\texttt{CTL}}[pid_i] = \perp$:
                send $\texttt{InitSign}$ to $(pid_i, (pid_i, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$                    {Generate CTL's signing keys
                wait for $(\texttt{InitSign}, \texttt{success}, pk_i)$
                $\mathsf{pk}_{\texttt{CTL}}[pid_i] \leftarrow pk_i$                    {Store public key
        if $\mathsf{pk}_{\texttt{CA}}[\mathsf{pid}_{\mathsf{cur}}] \neq \perp$:                    {Establish session at $\mathcal{F}_{\text{sig}}$ if it is not recorded
            send $\texttt{InitSign}$ to $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$                    {Generate CA's signing keys
            wait for $(\texttt{InitSign}, \texttt{success}, pk_{\texttt{CA}})$
            $\mathsf{pk}_{\texttt{CA}}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow pk_{\texttt{CA}}$                    {Store generated public key
        $\mathsf{CTLs}[pid_{\texttt{CA}}] \leftarrow (pid_1, \ldots, pid_n)$                    {Store CA's CTLs
        reply $(\texttt{InitMe}_{\texttt{CA}}, \mathsf{pk}_{\texttt{CA}}[\mathsf{pid}_{\mathsf{cur}}], (pid_1, \ldots, pid_n), (\mathsf{pk}_{\texttt{CTL}}[pid_1], \ldots, \mathsf{pk}_{\texttt{CTL}}[pid_n]))$
                    {Output CA's and CTLs' pubkeys to CA
    recv $\texttt{InitMe}_{\texttt{CTL}}$ from $\texttt{I/O}$:                    {CTL request initialization
        if $\mathbf{corr}(\mathsf{pid}_{\mathsf{call}}, (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$:
            reply $(\texttt{InitMe}_{\texttt{CTL}}, \perp)$                    {CTL key is corrupted, thus this entity cannot act as (trusted) CTL
        send responsively $\texttt{InitMe}_{\texttt{CTL}}$ to $\texttt{NET}$ $(\star)$                    {Query $\mathcal{A}$ for initialization details of $\mathsf{pid}_{\mathsf{cur}}$
        wait for $(\texttt{InitMe}_{\texttt{CTL}}, pid_1, \ldots, pid_n)$
        if $n < 1$:
            go to $(\star)$
        if $\mathsf{pk}_{\texttt{CTL}}[\mathsf{pid}_{\mathsf{cur}}] \neq \perp$:                    {Establish session at $\mathcal{F}_{\text{sig}}$ if it is not recorded
            send $\texttt{InitSign}$ to $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$                    {Generate CTL's signing keys
            wait for $(\texttt{InitSign}, \texttt{success}, pk_{\texttt{CTL}})$
            $\mathsf{pk}_{\texttt{CTL}}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow pk_{\texttt{CTL}}$                    {Store generated public key
        for all $i \in \{1, \ldots, n\}$ do:
            if $\mathsf{pk}_{\texttt{CA}}[pid_i] = \perp$:
                send $\texttt{InitSign}$ to $(pid_i, (pid_i, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{signer})$                    {Generate CA's signing keys
                wait for $(\texttt{InitSign}, \texttt{success}, pk_i)$
                $\mathsf{pk}_{\texttt{CA}}[pid_i] \leftarrow pk_i$                    {Store public key
        reply $(\texttt{InitMe}_{\texttt{CTL}}, \mathsf{pk}_{\texttt{CTL}}[\mathsf{pid}_{\mathsf{cur}}], (pid_1, \ldots, pid_n), (\mathsf{pk}_{\texttt{CA}}[pid_1], \ldots, \mathsf{pk}_{\texttt{CA}}[pid_n]))$
                    {Output CA's and CTL's pubkey to CA
    recv $\texttt{InitMe}_{\texttt{judge}}$ from $\texttt{I/O}$:                    {$\mathcal{P}_{\text{judge}}^{\texttt{CA}}$ requests initialization
        reply $(\texttt{InitMe}_{\texttt{judgeCA}}, \mathsf{pid}_{\texttt{CA}}, \mathsf{pk}_{\texttt{CA}}, \mathsf{pid}_{\texttt{CTL}}, \mathsf{pk}_{\texttt{CTL}}, \mathsf{CTLs})$                    {Output current state to $\mathcal{F}_{\text{judge}}$
    recv $(\texttt{GetCaCtls}, pid_{\texttt{CA}})$ from $\texttt{I/O}$:                    {Clients can query for CA/CTL pk's
        require: $pid_{\texttt{CA}} \in \mathsf{pid}_{\texttt{CA}} \land \mathsf{pk}[pid_{\texttt{CA}}] \neq \perp$
        Let $pid_1, \ldots, pid_n$ be the pids from $\mathsf{CTLs}[pid_{\texttt{CA}}]$
        reply $(\texttt{GetCaCtls}, \mathsf{pk}_{\texttt{CA}}[pid_{\texttt{CA}}], (pid_1, \ldots, pid_n), (\mathsf{pk}_{\texttt{CTL}}[pid_1], \ldots, \mathsf{pk}_{\texttt{CTL}}[pid_n]))$
    recv $\texttt{GetCAs}$ from $\texttt{I/O}$:                    {Clients can query for CAs and their pk's
        reply $(\texttt{GetCaCtls}, \mathsf{pid}_{\texttt{CA}}, \mathsf{pk}_{\texttt{CA}})$
    recv $\texttt{GetCTLs}$ from $\texttt{I/O}$:                    {Clients can query for CTLs and their pk's
        reply $(\texttt{GetCTLs}, \mathsf{pid}_{\texttt{CTL}}, \mathsf{pk}_{\texttt{CTL}})$
    recv $\texttt{initPartner}$ from $(pid, sid, role)$:                    {Request to initialize accompanied local judge or client
        $\mathsf{caller} \leftarrow (pid, sid, role)$
        if $role = \texttt{client}$:
            $\mathbf{init}((\texttt{local}, pid, \texttt{client}), sid, \texttt{judge})$                    {Trigger initialization of the entities local judge
        else if $role = \texttt{judge}$:
            Parse $pid$ as $(\texttt{local}, pid', \texttt{client})$
            $\mathbf{init}(pid', sid, \texttt{client})$                    {Trigger initialization of the accompanied client
        recv $\texttt{InitEntityDone}$ from $(pid, sid, role)$:                    {Finish initialization of accompanied local judge or client
            $caller \leftarrow \mathsf{caller}; \mathsf{caller} \leftarrow \perp$
            send $\texttt{initPartner}$ to $caller$
```

Fig. 26: The initialization ITM $\mathcal{F}_{\text{init}}^{\texttt{CA}}$.

Description of the certificate monitoring enforcing network functionality $\mathcal{F}_{\text{psync-net}} = (\texttt{psync-net})$:

```
Participating roles: {psync-net}
Corruption model: incorruptible
Protocol parameters:
    – $\delta \in \mathbb{N}$                                        {The upper bound for liveness
```

Description of $M^{\text{m}}_{\texttt{psync-net}}$:

```
Implemented role(s): {psync-net}
Subroutines: $\mathcal{P}^{\texttt{CA}}_{\text{client}}$ : client, $\mathcal{P}_{\texttt{CTL}}$ : ctl
Internal state:
    – $\tau \in \mathbb{N}, \tau = 0$                                  {Current time in the $\mathcal{F}_{\text{psync-net}}$
    – $\mathsf{counter} \in \mathbb{N}, \mathsf{counter} = 0$          {Request counter
    – $\mathsf{requestQueue} \subset \mathbb{N} \times \mathbb{N} \times (\{0,1\}^*)^4 \times \mathbb{N}^2, \mathsf{requestQueue} = \emptyset$
          {The set of messages to be delivered via $\mathcal{F}_{\text{psync-net}}$ of
           form $(ctr, t_{rec}, pid_s, pid_r, msg, resp, t_{del}, t_{out})$
    – $\mathsf{monReg} : \{0,1\}^* \to \mathbb{N}$                     {Last recorded check request of party $pid$. Initially all set to 0.
    – $\mathsf{pid}_{\texttt{CTL}} \subset \{0,1\}^*, \mathsf{pid}_{\texttt{CTL}} = \emptyset$   {The available CTLs
    – $\mathsf{pid}^{h}_{\mathsf{client}} \subset \{0,1\}^*$           {The set of honest clients
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
MessagePreprocessing:
        send GetCTLs to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\texttt{CA}}_{\text{init}} : \texttt{init})$   {Request CTLs at $\mathcal{F}^{\texttt{CA}}_{\text{init}}$
        wait for $(\texttt{GetCTLs}, CTL, pid_{\texttt{CTL}})$
        $\mathsf{pid}_{\texttt{CTL}} \leftarrow CTL$                   {Update available CTLs
        for all $pid \in \mathsf{pid}^{h}_{\mathsf{client}}$ do:
            send CorruptionStatus? to $(pid, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\texttt{CA}}_{\text{client}} : \texttt{client})$
            wait for $(\texttt{CorruptionStatus?}, b)$
            if $b$:
                $\mathsf{pid}^{h}_{\mathsf{client}}.\texttt{remove}(pid)$     {Remove corrupted party
                Remove entries for $pid$ from requestQueue and monReg.
Main:
        recv UpdateRound from NET:                                      {Triggering a clock update increases the time
            if $\exists(ctr, t_{rec}, pid_s, pid_r, msg, resp', t_{del}, t_{out}) \in \mathsf{requestQueue}$, s.t. $t_{del} = \bot \land \mathsf{round} - t_{rec} > \delta$:
                reply (UpdateRound, nack)                               {Requests need to delivered in $\delta$ time units to a CTL

            else if $\exists(ctr, t_{rec}, pid_s, pid_r, msg, resp', t_{del}, t_{out}) \in \mathsf{requestQueue}$, s.t. $t_{del} \neq \bot \land t_{rec} \neq \bot \land t_{out} = \bot \land \mathsf{round} - t_{del} > \delta$:
                reply (UpdateRound, nack)                               {Requests need to delivered in $\delta$ time units to a CTL

            else if $\exists pid$ in monReg s.t. $\mathsf{round} - \mathsf{monReg}[pid] > \delta$:
                reply (UpdateRound, nack)                               {$\mathcal{A}$ needs to allow honest parties to frequently trigger monitoring
            else:
                $\tau \leftarrow \tau + 1$                              {Clock update successful
        recv GetCurRound:                                               {Handling reads from the clock
            reply $(\texttt{GetCurRound}, \tau)$
        recv $(\texttt{Monitor}, pid, \{pid_1, \ldots, pid_l\})$ from I/O s.t. $\mathsf{pid}_{\mathsf{call}} = pid$, $\mathsf{role}_{\mathsf{call}} = \texttt{client}$, $\{pid_1, \ldots \land pid_l\} \subset \mathsf{pid}_{\texttt{CTL}}$: {Monitor CTLs
            for all $pid' \in \{pid_1, \ldots, pid_l\}$ do:               {Record monitor requests
                $\mathsf{counter} \leftarrow \mathsf{counter} + 1$
                $\mathsf{requestQueue}.\texttt{add}([\mathsf{counter}, \mathsf{round}, pid, pid', (\texttt{Monitor}, \mathsf{counter}, pid), \bot, \bot, \bot, \bot])$   {Record requests to CTLs
                $\mathsf{monReg}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow \mathsf{round}$     {Record last request of party
            send $(\texttt{Monitor}, \mathsf{requestQueue})$ to NET          {Leak full information to $\mathcal{A}$
        recv $(\texttt{Deliver}, ctr)$ from NET:     {$\mathcal{A}$ triggers delivery of message with id ctr to $\mathcal{F}_{\texttt{CTL}}$ or $\mathcal{P}^{\texttt{CA}}_{\text{client}}$
            if $\exists(ctr, t_{rec}, pid_s, pid_r, msg, resp, t_{del}, t_{out}) \in \mathsf{requestQueue}$, s.t. $t_{del} = \bot$:
                $\mathsf{requestQueue}.\texttt{remove}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp, t_{del}, t_{out}])$
                $\mathsf{requestQueue}.\texttt{add}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp, \mathsf{round}, t_{out}])$   {Record delivery to CTL
                send $(msg, ctr)$ to $(pid_r, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\texttt{CTL}} : \texttt{ctl})$   {Deliver request to CTL
            else if $\exists(ctr, t_{rec}, pid_s, pid_r, msg, resp, t_{del}, t_{out}) \in \mathsf{requestQueue}$, s.t. $t_{del} \neq \bot \land t_{out} = \bot$:
                $\mathsf{requestQueue}.\texttt{remove}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp, t_{del}, t_{out}])$
                $\mathsf{requestQueue}.\texttt{add}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp, t_{del}, \mathsf{round}])$   {Record delivery to client
                send $(resp)$ to $(pid_s, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\texttt{CA}}_{\text{client}} : \texttt{client})$   {Deliver answer to client
        recv $(\texttt{Monitor}, ctr, resp)$ from I/O:                      {Response from CTL on monitor request
            if $\exists(ctr, t_{rec}, pid_s, pid_r, msg, resp', t_{del}, t_{out}) \in \mathsf{requestQueue}$, s.t. $t_{out} = \bot$:
                $\mathsf{requestQueue}.\texttt{remove}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp', t_{del}, t_{out}])$
                $\mathsf{requestQueue}.\texttt{add}([ctr, t_{rec}, pid_{\mathsf{s}}, pid_{\mathsf{r}}, msg, resp, t_{del}, t_{out}])$   {Record reponse of CTL
            send Monitor, requestQueue to NET                               {Leak full information to $\mathcal{A}$
        recv Establish from I/O:                                            {Client registers at $\mathcal{F}_{\text{psync-net}}$
            $\mathsf{pid}_{\mathsf{client}}.\texttt{add}(\mathsf{pid}_{\mathsf{call}})$
            $\mathsf{monReg}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow \mathsf{round}$      {Record registration round as starting point in monReg
```

Fig. 27: The certificate monitoring enforcing network functionality $\mathcal{F}_{\text{psync-net}}$ for modeling a partially synchronous network.

Description of $\mathcal{P}^{\texttt{CA}}_{\text{judge}} = (\texttt{judge})$:

Participating roles: $\{\texttt{judge}\}$
Corruption model: *Dynamic corruption without secure erasures*

Description of $M_{\texttt{judge}}$:

```
Implemented role(s): {judge}
Subroutines: $\mathcal{F}_{\text{sig}}$ : verifier, $\mathcal{F}^{\texttt{CA}}_{\text{init}}$ : init
Internal state:
  – $\mathsf{pid}_{\texttt{CA}} \subset \{0,1\}^*$                                   {The pids of the CAs
  – $\mathsf{pk}_{\texttt{CA}} : \mathsf{pid}_{\texttt{CA}} \to \{0,1\}^*$            {The public keys of CAs, initially ⊥ for all entries
  – $\mathsf{pid}_{\texttt{CTL}} \subset \{0,1\}^*$                                  {The pids of the CTLs
  – $\mathsf{pk}_{\texttt{CTL}} : \mathsf{pid}_{\texttt{CTL}} \to \{0,1\}^*$          {The public keys of CTLs, initially ⊥ for all entries
  – $\mathsf{CTLs} : \mathsf{pid}_{\texttt{CA}} \to (\mathsf{pid}_{\texttt{CTL}} \cup \{\bot\})^*$   {A CAs trusted CTLs, initally ⊥
  – $\mathsf{verdicts} \in \{0,1\}^*, \mathsf{verdicts} = \varepsilon$               {Recorded verdict
CheckID($pid, sid, role$):
  Accept all messages with the same $sid$ addressed to $(pid, sid, \texttt{judge})$ where $pid$ can be parsed as $(\texttt{local}, pid', \texttt{client})$.
Corruption behavior:
  – AllowCorruption($pid, sid, role$) :
      Parse $\mathsf{pid}_{\mathsf{cur}}$ as $(pid', role')$
      $corrRes \leftarrow \mathbf{corr}(pid', \mathsf{sid}_{\mathsf{cur}}, role')$
      return $corrRes$                                 {Corruption is allowed if accompanied client is corrupted.
  – DetermineCorrStatus($pid, sid, role$) :          {Consider local judge corrupted if associated client is corrupted.
      Parse $pid$ as $(pid', role')$
      $corr \leftarrow \mathbf{corr}(pid', sid, role')$
      if $corr$:
          return true
      else:
          return false
EntityInitialization:
    if ITM was not activated via init macro, resp. a InitEntity message:
        send initPartner to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\texttt{CA}}_{\text{init}} : \texttt{init})$
        wait for initPartner
MessagePreprocessing:
    send $\texttt{InitMe}_{\texttt{judge}}$ to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\texttt{CA}}_{\text{init}} : \texttt{init})$     {Update CA/CTL identities and pubkeys
    wait for $(\texttt{InitMe}_{\texttt{judge}}, pid_{\texttt{CA}}, pk_{\texttt{CA}}, pid_{\texttt{CTL}}, pk_{\texttt{CTL}}, CTLs)$
    $\mathsf{pid}_{\texttt{CA}} \leftarrow pid_{\texttt{CA}}$; $\mathsf{pk}_{\texttt{CA}} \leftarrow pk_{\texttt{CA}}$; $\mathsf{pid}_{\texttt{CTL}} \leftarrow pid_{\texttt{CTL}}$; $\mathsf{pk}_{\texttt{CTL}} \leftarrow pk_{\texttt{CTL}}$; $\mathsf{CTLs} \leftarrow CTLs$
Main:
    recv $(\mathsf{Evidence}_M, (ctr', [serial, pid, pk, pid'_{\texttt{CA}}, \sigma_{\texttt{CA}}], pid'_{\texttt{CTL}}, \tau, \sigma_{\texttt{CTL}}))$ from I/O     {Client complains that someone registered
        s.t. $\mathsf{pid}_{\mathsf{call}} = \mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{call}} = \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{call}} = \texttt{client}$:     a certificate for her identity
        $b \leftarrow \texttt{verifySig}([(ctr', [serial, \mathsf{pid}_{\mathsf{cur}}, pk, pid'_{\texttt{CA}}, \sigma_{\texttt{CA}}], pid_{\texttt{CTL}}, \tau], \sigma_{\texttt{CTL}}, pid_{\texttt{CTL}}[pid'_{\texttt{CTL}}], pid'_{\texttt{CTL}})$   {Check CTL signature
        if $pid'_{\texttt{CA}} \in \mathsf{pid}_{\texttt{CA}}$:
            $c \leftarrow \texttt{verifySig}([serial, \mathsf{pid}_{\mathsf{cur}}, pk, pid'_{\texttt{CA}}], \sigma_{\texttt{CA}}, pid_{\texttt{CA}}[pid'_{\texttt{CA}}], pid'_{\texttt{CA}})$     {Check CA signature
        else:
            $c \leftarrow \texttt{false}$
        if $b \wedge c$:                                                      {Certificate is valid
            $\mathsf{verdicts.add}(\mathsf{dis}(pid_{\texttt{CA}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\texttt{CA}} : \texttt{ca}))$       {The local judge blames the CA for misbehavior
    recv GetVerdict:                                                   {Both I/O and NET may send this message.
        reply $(\texttt{GetVerdict}, \mathsf{verdicts})$
    recv $(\texttt{GetJudicialReport}, msg)$ from I/O:
        reply $(\texttt{GetJudicialReport}, \varepsilon)$
Procedures and Functions:
    function verifySig($msg, \sigma, pk, pid$) :                        {Verify signature at $\mathcal{F}_{\text{sig}}$
        send $(\texttt{VerResult}, msg, \sigma, pk)$ to $((\mathsf{pid}_{\mathsf{cur}}, \texttt{ctl}), (pid, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \texttt{verifier})$
        wait for $(\texttt{VerResult}, result)$
        return $result$
```

Fig. 28: The judging functionality $\mathcal{P}^{\texttt{CA}}_{\text{judge}}$ for a CA (Part 1).

Description of $\mathcal{P}^{\texttt{CA}}_{\text{sv}} = (\texttt{supervisor})$:

```
Participating roles: {supervisor}
Corruption model:  incorruptible
Protocol parameters:
    – $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$                {The set of security properties that may break
```

Description of $M^{\texttt{CA}}_{\texttt{supervisor}}$:

```
Implemented role(s): {supervisor}
Subroutines: $\mathcal{P}^{\texttt{CA}}$ : ca, $\mathcal{P}_{\text{CTL}}$ : ctl
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
        recv (IsAssumptionBroken?, $msg$):
            reply (IsAssumptionBroken?, false)
        recv (corruptInt?, ($pid, sid, role$)) s.t. $role \notin$ {client, judge, supervisor}:
            if $role \neq$ ca:
                reply (corruptInt, false)
            else:
                $corrRes \leftarrow \mathbf{corr}(pid, \mathsf{sid}_{\mathsf{cur}}, \texttt{ca})$        {Request corruption status at $\mathcal{P}^{\texttt{CA}}$
                reply (corruptInt, $corrRes$)
```

Fig. 29: The supervisor $\mathcal{P}^{\texttt{CA}}_{\text{sv}}$ for CA.

Local judges

When an *uncorrupted* instance $((pid, role), sid, \texttt{judge})$ of $\mathcal{P}^{\texttt{CA}}_{\text{judge}}$ in $\mathcal{R}'$ renders a verdict, i. e., verdicts $\neq \varepsilon$, or its verdicts changes, $\mathcal{S}$ extracts the verdicts and forwards it to $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. More specifically, $\mathcal{S}$ sends ($\texttt{BreakAccProp}$, $verdict$, $\{(\texttt{correctCert}, (pid, sid, role))\}$) to $((\texttt{local}, pid, role), sid, \mathcal{F}^{\text{acc}}_{\text{PKI}} : \texttt{judge})$ where $verdict$ is a map from $\mathsf{ids}_{\mathsf{assumption}} \to \{0,1\}^*$ where the entry $verdict[id] =$ verdicts (contains the verdict from $\mathcal{P}^{\texttt{CA}}_{\text{judge}}$). All other entries in $verdict$ are mapped to ε.

If a judge's associated client is corrupted, the local judge is also considered corrupted and serves as pure message forwarder to/from $\mathcal{A}$. Thus – as stated above – $\mathcal{S}$ forwards the messages from/to $\mathcal{A}$ to/from $\mathcal{I}$ in this case.

This concludes the description of the simulator. It is easy to see that *(i)* $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded, and *(ii)* $\mathcal{S}$ is a responsive simulator for $\mathcal{I}$, i. e., restricting messages from $\mathcal{I}$ are answered immediately as long as $\{\mathcal{S}, \mathcal{I}\}$ runs with a responsive environment. We now argue that $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$.

In the following part of the proof, let $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$ be an arbitrary but fixed environment. First, observe that $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ provides $\mathcal{S}$ with full information about *(i)* all creations requests for certificate of uncorrupted clients (for which also $\texttt{correctCert}$ holds true) and *(ii)* retrieve requests performed by higher-level protocols/the environment. Hence, $\mathcal{S}$'s simulated protocol $\mathcal{R}'$ obtains the same inputs as $\mathcal{R}$ (resp. $\mathcal{I}$) and thus performs identical to $\mathcal{R}$. We remark that further I/O input, e. g., requests to judges, do not influence the state of $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. Thus, we can conclude that the network behavior simulated by $\mathcal{S}$ towards the environment is indistinguishable from the network behavior of $\mathcal{R}$. Moreover, we can also conclude that the corruption status of (internal) entities in the real and ideal world is synchronized. Since the simulator has full control over corrupted entities, which are handled via the internal simulation $\mathcal{R}'$, this implies that the I/O behavior of corrupted entities of $\mathcal{R}/\mathcal{I}$ towards the environment is also identical in the real and ideal world. Note that purely internal (private) parties have no interface to the environment. The only way to potentially distinguish the real and ideal world is the I/O behavior of honest entities of $\mathcal{R}/\mathcal{I}$ towards higher-level protocols.

We will now go over all possible interactions with honest entities on the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the environment, i. e., are also indistinguishable. At the start of a run, there were no interactions on the I/O interface with honest parties yet. Thus, $\mathcal{R}$ and $\mathcal{I}$ are indistinguishable. In the following, we assume that all I/O interactions so far have resulted in the same behavior visible towards the environment in both the real and ideal world.

In what follows, we call the state which includes the certificates of uncorrupted parties for which $\texttt{correctCert}$ still holds true *relevant state*. The relevant state can be extracted from the union of all state variables of all $\mathcal{P}_{\text{CTL}}$ instances in a session of $\mathcal{P}^{\text{acc}}_{\text{PKI}}$ in $\mathcal{R}$, resp. $\mathcal{R}'$, by removing certificates from *(i)* corrupted parties and *(ii)* from honest parties whose local judges already store a verdict$\neq \varepsilon$. In $\mathcal{I}$, the relevant state can be derived from $\mathcal{F}^{\text{acc}}_{\text{PKI}}$'s

Description of the ideal authenticated channel functionality $\mathcal{F}_{\texttt{auth}} = (\texttt{auth})$:

```
Participating roles: {auth}
Corruption model: custom
```

Description of $M_{\texttt{auth}}$:

```
Implemented role(s): {auth}
Internal state:
  – queue : $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$    {Queue of messages from entity $e_1$ to entity $e_2$, initially $\perp$ for all entries
  – status : $(\{0,1\}^*)^3 \to \{\texttt{inactive}, \texttt{active}, \texttt{established}\}$    {The status of $(pid, sid, role)$, initially inactive for all entries
  – corrStatus $\in \{0,1\}$    {The corription status of the $\mathcal{F}_{\texttt{auth}}$
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Main:
        recv corrupt from NET:    {$\mathcal{A}$ (tries to) corrupts current instance of $\mathcal{F}_{\texttt{auth}}$
          if $\forall$ entries in status are not equal to established:    {This models static corruption
            corrStatus $\leftarrow$ true
            reply (corrupt, ack)
          else:
            reply (corrupt, nack)
        recv CorruptionStatus? from I/O:    {Allows environment to check correct simulation of corrupted parties.
          if corrStatus = true:
            reply (CorruptionStatus?, true)
          else:
            reply (CorruptionStatus?, false)
        recv (Establish, $m$) from I/O s.t. $\mathsf{sid_{call}} = \mathsf{sid_{cur}}$, status $[(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})]$ = inactive:    {Establish session
          status$[(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})] \leftarrow$ active
          send (Establish, $m$, $(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$) to NET
        recv (Establish, $(pid, \mathsf{sid_{cur}}, role)$) from NET s.t. status $[(pid, \mathsf{sid_{cur}}, role)]$ = active:    {Establish session
          status$[(pid, \mathsf{sid_{cur}}, role)] \leftarrow$ established
          send (Establish, $m$, $(pid, \mathsf{sid_{cur}}, role)$) to $(pid, \mathsf{sid_{cur}}, role)$
        recv (Send, $(pid, \mathsf{sid_{cur}}, role)$, $msg$) from I/O s.t. status $[(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})]$ = established $\wedge\, pid \neq \mathsf{pid_{call}}$ :    {Send message via authenticated channel
          if corrStatus = false:
            queue$[(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}}), (pid, \mathsf{sid_{cur}}, role)]$.add($msg$)    {Add $msg$ to the queue of $pid$
          send (Send, $(\mathsf{pid_{call}}, \mathsf{sid_{call}}, \mathsf{role_{call}})$, $(pid, \mathsf{sid_{cur}}, role)$, $msg$) to NET    {Leak communication to NET
        recv (Deliver, $(pid_1, \mathsf{sid_{cur}}, role_1)$, $(pid_2, \mathsf{sid_{cur}}, role_2)$, $msg$) from I/O
         s.t. status $[(pid_2, \mathsf{sid_{cur}}, role_2)]$ = established $\wedge\, pid_1 \neq pid_2$ :    {Deliver message via authenticated channel
          if corrStatus = false:
            if queue[, $(pid_1, \mathsf{sid_{cur}}, role_1)$, $(pid_2, \mathsf{sid_{cur}}, role_2)$] = $\perp$:    {There are no queued messages
              reply (Deliver, $\perp$)    {Return error
            else:
              remove the first message from queue$[(pid_1, \mathsf{sid_{cur}}, role_1), (pid_2, \mathsf{sid_{cur}}, role_2)$, let $msg'$ be this message

              send (Received, $(pid_1, \mathsf{sid_{cur}}, role_1)$, $msg'$) to $(pid_2, \mathsf{sid_{cur}}, role_2)$    {Deliver first message from queue
          else:
            send (Received, $(pid_1, \mathsf{sid_{cur}}, role_1)$, $msg$) to $(pid_2, \mathsf{sid_{cur}}, role_2)$    {Deliver message from adversary if corrupted
        recv (Drop, $(pid_1, \mathsf{sid_{cur}}, role_1)$, $(pid_2, \mathsf{sid_{cur}}, role_2)$) from NET:
          if queue$[(pid_1, \mathsf{sid_{cur}}, role_1), (pid_2, \mathsf{sid_{cur}}, role_2)] \neq \perp$:
            remove the first message from queue$[(pid_1, \mathsf{sid_{cur}}, role_1), (pid_2, \mathsf{sid_{cur}}, role_2)]$.    {Drop message
            reply (Drop, ack)
```

Fig. 30: The ideal authenticated channel functionality $\mathcal{F}_{\mathrm{auth}}$ (cf. [70].

state variable by excluding *(i)* certificates of corrupted parties and *(ii)* certificates where correctCert does not hold true for the certificate owner.

Certificate creation/registration requests: Certificate creation/registration requests do not directly result in an output to the environment. But, they might affect the output of $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ later on as they have direct impact on the relevant state of $\mathcal{I}/\mathcal{R}'$, resp. $\mathcal{R}$. Thus, we now show that registration requests behave "identical", i. e., we have to argue that these changes in relevant state are "synchronized" between $\mathcal{I}$ and $\mathcal{R}'$. In particular, the buffered set of relevant certificates registrations, i. e., certificate requested by uncorrupted parties for which correctCert still holds true, in $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ is equal to the buffered set of certificate registration from the same class of clients in $\mathcal{R}'$. We define *buffered registration request* in $\mathcal{R}'$ as the following requests/messages: *(i)* certificate requests stored in uncorrupted instances of $\mathcal{P}_{\mathrm{client}}^{\mathtt{CA}}$ (in pk and certCA) for which correctCert still holds true *(ii)* minus the set of finalized certificates from the different instances of $\mathcal{P}_{\mathtt{CTL}}$. In what follows, we will call these certificates the *buffered relevant certificates for registration at* $\mathcal{R}'$.

Observe that, upon receiving a certificate registration, $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ behaves similar to $\mathcal{R}'$: in $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$, the submitted certificate registration is directly stored in $\mathsf{buffer_S}$. In $\mathcal{R}'$, the submit message is stored in $\mathcal{P}_{\mathrm{client}}^{\mathtt{CA}}$ and forwarded

Description of the protocol $\mathcal{F}_{\text{sig}} = (\mathtt{signer}, \mathtt{verifier})$:

```
Participating roles: $\{\mathtt{signer}, \mathtt{verifier}\}$
Subroutines: $\mathcal{F}_{\mathtt{init}}^{\mathsf{CA}} : \mathtt{init}$
Corruption model: dynamic with secure erasures
Protocol parameters:
    – $\mathsf{p} \in \mathbb{Z}[x]$.        {Polynomial that bounds the runtime of the algorithms provided by the adversary.
```

Description of $M_{\mathtt{signer},\mathtt{verifier}}$:

```
Implemented role(s): $\{\mathtt{signer}, \mathtt{verifier}\}$
Internal state:
    – $(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^4 = (\bot, \bot, \bot, \bot)$.        {Algorithms and key pair.
    – $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$.        {Party ID of the key owner.
    – $\mathsf{msglist} \subset \{0,1\}^* = \emptyset$.        {Set of recorded messages.
    – $\mathsf{KeysGenerated} \in \{\mathtt{ready}, \bot\} = \bot$.        {Has signer initialized his key?
CheckID$(pid, sid, role)$:
    Check that $sid = (pid', sid')$:
    If this check fails, output reject.
    Otherwise, accept all entities with the same SID.        {A single instance manages all parties and roles in a single session.
Corruption behavior:
    – AllowCorruption$(pid, sid, role)$:        {The highlighted part differs between a standard version of $\mathcal{F}_{\text{sig}}$ and the version considered here
        Parse $\mathsf{pid}_{\mathsf{cur}}$ to $(pid', role')$
        send GetCTLs to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathtt{init}}^{\mathsf{CA}} : \mathtt{init})$        {Query for the list of CTLs of session sid, decline corruption requests for CTL keys
        wait for $(\mathtt{GetCaCtls}, pid_{\mathtt{CTL}}, pk_{\mathtt{CTL}})$
        if $pid' \in pid_{\mathtt{CTL}} \land role' = \mathtt{ctl}$:
            return false        {Not allowed to corrupt CLT keys.
        else if $role = \mathtt{verifier}$:        {Corruption of verifier instances not allowed
            return false
        else:
            return true
    – LeakedData$(pid, sid, role)$: If $(pid, sid, role)$ determines its initial corruption status, use the default behavior of LeakedData.
      Otherwise, if $role = \mathtt{signer}$ and $pid = \mathsf{pidowner}$, return KeysGenerated. In all other cases return $\bot$.
Initialization:
        send responsively InitMe to NET;
        wait for $(\mathtt{Init}, (sig, ver, pk, sk))$.
        $(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \leftarrow (sig, ver, pk, sk)$.
        Parse $\mathsf{sid}_{\mathsf{cur}}$ as $(pid, sid)$.
        $\mathsf{pidowner} \leftarrow pid$.
Main:
        recv InitSign from I/O to $(\mathsf{pidowner}, \_, \mathtt{signer})$:        {Successful initialization. Note that signer can submit InitSign multiple times, always with the same effect.
            $\mathsf{KeysGenerated} \leftarrow \mathtt{ready}$.
            reply $(\mathtt{InitSign}, \mathtt{success}, \mathsf{pk})$.
        recv $(\mathtt{Sign}, msg)$ from I/O to $(\mathsf{pidowner}, \_, \mathtt{signer})$ s.t. $\mathsf{KeysGenerated} = \mathtt{ready}$:
            $\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
            $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$.        {Sign and check that verification succeeds.
            if $\sigma = \bot \lor b \neq \mathtt{true}$:
                reply $(\mathtt{Signature}, \bot)$.        {Signing or verification test failed.
            else:
                add $msg$ to $\mathsf{msglist}$.
                reply $(\mathtt{Signature}, \sigma)$.        {Record $msg$ for verification and return signature.
        recv $(\mathtt{Verify}, msg, \sigma, pk)$ from I/O to $(\_, \_, \mathtt{verifier})$:
            $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, pk)$.        {Verify signature.
            if $pk = \mathsf{pk} \land b = \mathtt{true} \land msg \notin \mathsf{msglist} \land (\mathsf{pidowner}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{signer}) \notin \mathsf{CorruptionSet}$:
                reply $(\mathtt{VerResult}, \mathtt{false})$.        {Prevent forgery.
            else:
                reply $(\mathtt{VerResult}, b)$.        {Return verification result.
```

Fig. 31: The ideal signature functionality $\mathcal{F}_{\text{sig}}$.

via $\mathcal{F}_{\text{auth}}$ to $\mathcal{P}^{\texttt{CA}}$. So, the buffered relevant certificates $\mathcal{R}'$ also are in a buffer of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$, i. e., the buffered relevant certificates are synchronized between $\mathcal{R}'$ and $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ (that is, the sets are equal in both worlds). Note that $\mathcal{S}$ does not have to take care of certificates of corrupted parties as $\mathcal{A}/\mathcal{S}$ fully determines the output of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ for these parties anyways. Thus, $\mathcal{S}$ is indeed able to keep the buffered relevant certificates in $\mathcal{I}$ and $\mathcal{R}'$ synchronized.

State updates: Again, state updates do not lead to a direct output to the environment. However, state updates may influence the output to the environment later on. Thus, we argue here that the relevant state (see above) is synchronized between $\mathcal{I}$ and $\mathcal{R}'$ (thus, it will allow for the same output in $\mathcal{I}$ and $\mathcal{R}$). As the buffered relevant certificates in $\mathcal{R}'$ and $\mathcal{I}$ are synchronized (see above), we can conclude that the relevant states of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ and $\mathcal{R}'$ stay synchronized during state updates. This boils down to mirroring the relevant state (see above for the definition) from the CTLs to $\mathcal{I}$ (as $\mathcal{S}$ does). Thus, all relevant certificates which $\mathcal{S}$ would add to the $\mathtt{state}$ in $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ are also buffered in $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. Therefore, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ will accept every state update of $\mathcal{S}$ and thus the relevant state stay synchronized in $\mathcal{I}$ and $\mathcal{R}'$.

Retrieve requests: Observe that, upon a retrieve request, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ stores the read request in $\mathsf{buffer}_{\mathsf{R}}$ and leaks the full request to the simulator including an ID and the receiver of the response. Note that the procedure is analogously to a certificate registration. A retrieve request does not directly result in an output to the environment. With the same argumentation as above follows that $\mathcal{S}$ is able to keep the "retrieve buffers", i. e.the set of requested certificates stored in honest instances of $\mathcal{P}_{\text{client}}^{\texttt{CA}}$, resp. $\mathcal{F}_{\text{PKI}}^{\text{acc}}$, for honest participants of $\mathcal{I}$ and $\mathcal{R}'$ synchronized.
Deliver responses to retrieve requests: If $\mathcal{S}$ is triggered to output a response to a retrieve request, i. e., $\mathcal{S}$ receives `Retrieve` via NET including the necessary data to output, $\mathcal{S}$ firstly simulates received input in $\mathcal{R}'$ and determines whether this input actually leads to an output on an I/O interface. In case of an output, $\mathcal{S}$ forwards the output in a `Deliver` message to $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. In this case, there are two cases to consider: *(i)* `correctCert` is broken for the certificate owner pid or *(ii)* `correctCert` still hold true for pid. In the first case, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ will simply forward the input from $\mathcal{S}$ to the requestor as all checks at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ are disabled. As the output is extracted from $\mathcal{R}'$, we can conclude that $\mathcal{R}$ and $\mathcal{I}$ are indistinguishable. In the second case, the output is honestly generated in $\mathcal{R}'$ and $\mathcal{I}$. As argued above, the relevant states of both worlds are synchronized. Thus, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ will accept the `Deliver` command of $\mathcal{S}$ and the outputs in both worlds will be identical.

When a party queries for a certificate of corrupted party, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ allows $\mathcal{A}/\mathcal{S}$ to freely determine the certificate delivered to the requesting party. We already discussed this case after the description of the simulator and emphasize that in this case, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ simply forwards the output provided by $\mathcal{S}$ (which matches $\mathcal{R}'$).

Altogether, both worlds are also indistinguishable in the case of retrieve requests.

Monitoring Process: Note that the monitoring process only takes places in $\mathcal{R}'$ and only its output, a verdict if there is any, is mirrored to $\mathcal{I}$ (see below). Thus, the monitoring process will not cause a case where $\mathcal{I}$ and $\mathcal{R}$ are distinguishable.

Verdicts: First note, that $\mathcal{S}$ sends a `BreakAccProp` message to break `correctCert` when an unambiguous or an not requested certificate of an uncorrupted party in $\mathcal{R}'$ appears. More specifically, $\mathcal{S}$ sends a `BreakAccProp` when there is a new/updated verdict available in an uncorrupted instance of $\mathcal{P}_{\text{judge}}^{\texttt{CA}}$. We remark that these requests match the rules in $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$ and thus are always accepted by $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. Further, we assume that the used signature scheme is EUF-CMA secure. Thus, the probability that an honest party receives and accepts a certificate created by $\mathcal{A}$ via forging signatures (without corrupting the party owning the key or corrupting the key itself) only occurs with negligible probability. Also, the `verifier` instance of an honest local judge is incorruptible. Thus, we can conclude that $\mathcal{S}$ can always keep the verdicts and whether consistency is broken synchronized in $\mathcal{I}$ and $\mathcal{R}'$. Thus, the output on a `GetVerdict` request is equal in $\mathcal{R}'$ and $\mathcal{I}$. Thus, ideal and real world are indistinguishable.

We also remark that $\mathcal{P}_{\text{client}}^{\texttt{CA}}$ only forwards certificates to the environment if they are older than $3 \cdot \delta$ time units. In this case, the monitoring process for the particular certificate is already finished (in $\mathcal{R}/\mathcal{R}'$) and thus the associated local judge will provide a verdict in case of a maliciously created certificate. This triggers $\mathcal{S}$ to break `correctCert` (including providing a verdict) for the dedicated party and ensures that there is always a verdict available/the status of `correctCert` correctly reflected/synchronized between $\mathcal{R}'$ and $\mathcal{I}$. Thus, also in this special case, $\mathcal{R}$ and $\mathcal{I}$ are indistinguishable.

Judicial Reports: The judicial report (for uncorrupted instances of $\mathcal{F}_{\text{judgeParams}}^{\text{acc-PKI}}$) in both worlds is always ε. Thus, $\mathcal{R}$ and $\mathcal{I}$ stay indistinguishable.

Supervisor: The same holds true for `BreakAssumption` requests to the supervisor. The message of $\mathcal{S}$ matches the rules and thus will be accepted.

Altogether, $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ behave identical in terms of behavior visible to the environment $\mathcal{E}$ and thus are indistinguishable.

$\square$

E.4 Deterrence Analysis: As running CA business is profitable in practice and clients benefit from PKIs, we should have that $U^i_{hP} \gg 0$. We assume U^i_{hL} to be negligible as parties falsely accusing another cannot provide undeniable evidence that the party indeed misbehaved. As all relevant data is public in the PKI setting, we assume $U^i_{hE} = 0$. In real world scenarios, CAs would typically lose their complete reputation when misbehaving and thus often have to close business afterwards [92]. Thus, we estimate $U^c_l \ll U^i_{hP}$. In order to deter CAs from misbehavior, penalties need to be chosen significantly high such that $U^i_{mL} \gg U^i_{mP} - U^i_{hP}$. As it is not possible to generate faked undeniable evidence, we assume $U^i_{mE} = 0$. Thus, Equation 1 and 2 are met.

F. Key Exchange Based on $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ (Full Details)

Before we have a look at the security proof of Theorem 3, we present the ideal functionality $\mathcal{F}^{\text{acc}}_{\text{KE}}$ and the key exchange protocol specified in ISO 9798-3 [56] in detail. This case study relies on existing analysis of the ISO protocol [20, 27, 66]. More specifically, we derive the considered ideal key exchange functionality $\mathcal{F}_{\text{KE}}$ from the functionality explained in [20] and use AUC to derive $\mathcal{F}^{\text{acc}}_{\text{KE}}$ from $\mathcal{F}_{\text{KE}}$. In particular, we add the accountability property `authenticity` to $\mathcal{F}^{\text{acc}}_{\text{KE}}$. $\mathcal{F}^{\text{acc}}_{\text{KE}}$ ensures for "honest" key exchange sessions, that they provide security guarantees as long as non of the parties was involved in a misconduct in $\mathcal{F}^{\text{acc}}_{\text{PKI}}$. In contrast to existing works, which typically rely on an ideal/perfect PKI which does not allow maliciously generated certificates, our model of the ISO protocol $\mathcal{P}^{\text{acc}}_{\text{iso}}$ uses the PKI $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ for accessing the public signature keys of the involved parties for mutual authentication. Theorem 2 then allows us to replace $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ with its realization $\mathcal{P}^{\text{acc}}_{\text{PKI}}$, a realistic PKI with CTLs.

In this section, we will firstly provide the formal specification of the (accountable) ideal key exchange functionality $\mathcal{F}^{\text{acc}}_{\text{KE}}$ and the model of the ISO protocol $\mathcal{P}^{\text{acc}}_{\text{iso}}$. Finally, we will provide a formal proof for Theorem 3.

F.1 The Accountable Key Exchange Functionality $\mathcal{F}^{\text{acc}}_{\text{KE}}$: We also provide the full formal specification of $\mathcal{F}^{\text{acc}}_{\text{KE}}$ in Figure 32 and 33 including the specification of $\mathcal{F}^{\text{KE}}_{\text{judgeParams}}$ in Figure 34. The highlighted parts in the specification of $\mathcal{F}^{\text{acc}}_{\text{KE}}$ mark the changes during the AUC transformation from $\mathcal{F}_{\text{KE}}$ to $\mathcal{F}^{\text{acc}}_{\text{KE}}$. In what follows, we firstly present $\mathcal{F}_{\text{KE}}$ and then discuss the differences between $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}^{\text{acc}}_{\text{KE}}$.

The functionality $\mathcal{F}_{\text{KE}}$ consists of two roles, `initiator` and `responder`. Both, `initiator` and `responder`, are implemented via a single machine. One instance of this machine models a single key exchange. In $\mathcal{F}_{\text{KE}}$, key exchange sessions are predefined via SIDs of form (sid', pid_i, pid_r) where sid' is a local session ID for the key exchange, pid_i is the PID of the initiator and pid_r is the responder's PID. If two honest entities finish a key exchange, then $\mathcal{F}^{\text{acc}}_{\text{KE}}$ ensures that they obtain an ideal session key which is unknown to the adversary.

$\mathcal{F}_{\text{KE}}$ models a key exchange in three "phases" which are reflected in the available commands:

Public key registration (I/O): before parties can start a key exchange, they need to register their identities.

Initialization of key exchange (I/O): when parties have registered, they can start a key exchange.

Finish key exchange (Net): $\mathcal{F}_{\text{KE}}$ allows the adversary/simulator to determine when a started key exchange finished. The `FinishKE` message triggers the output of keys to higher-level protocols. In a session consisting of honest initiator and responder, the session key is derived ideally, i. e., chosen uniformly at random from a cyclic group (G, n, g). If one of the parties is corrupted, the adversary/the simulator may determine the session key.

In a session without corrupted parties, $\mathcal{F}_{\text{KE}}$ provides mutual authentication.

We add the following to $\mathcal{F}_{\text{KE}}$ to derive $\mathcal{F}^{\text{acc}}_{\text{KE}}$: if one of the two parties involved in a key exchange does not provide authenticity, i. e., the parties are not corrupted but $\text{brokenProps}[\texttt{authenticity}, (\texttt{local}, pid, role)] = \texttt{true}$ for one of both participants, $\mathcal{A}$ is allowed to determine the session key during key exchange. If $\mathcal{F}^{\text{acc}}_{\text{KE}}$ already established a session key, the key is leaked to $\mathcal{A}$. Additionally, $\mathcal{F}^{\text{acc}}_{\text{KE}}$, resp. $\mathcal{F}^{\text{KE}}_{\text{judgeParams}}$, restricts the local judges output to be a verdict blaming some internal protocol party and/or the intended key exchange partner. Note that the local judge considers itself as corrupted if the accompanied initiator, resp. responder entity is corrupted. Also, a local judge can only be (directly) corrupted if the accompanied initiator, resp. responder entity is corrupted. If the local judge is directly corrupted, it acts (as common) as pure message forwarder for $\mathcal{A}$. Judicial reports do not contain content, i. e., the judicial report is always ε. We do not consider a supervisor in this example/functionality. Formally, the supervisor does not accept break attempts and always outputs an empty leakage.

F.2 The ISO 9798-3 Protocol as Accountable Key Exchange Protocol: The ISO protocol [56] or in short the *ISO Protocol* as depicted in Figure 8 is a key exchange protocol which builds mutual authentication on top of a Diffie-Hellman key exchange.

We model the ISO protocol in a modular way using several smaller protocols. The static structure of all protocols, including their I/O connections for direct communication, is shown in Figure 35. In the following, we give a high-level overview over each part of the protocol.

The (accountable) ISO protocol is modeled as a real protocol $\mathcal{P}_{\text{iso}}^{\text{acc}}$. We derived $\mathcal{P}_{\text{iso}}^{\text{acc}}$ mainly from $\mathcal{P}_{\text{iso}}$ [17]. However, as already stated above, we model the key exchange via an pre-established SID which predetermines the session of the key exchange and initiator and responder in the particular session (cf., e. g., [20]). In what follows, we firstly explain our variant of $\mathcal{P}_{\text{iso}}$. Then, we explain how to derive the accountable variant $\mathcal{P}_{\text{iso}}^{\text{acc}}$ from $\mathcal{P}_{\text{iso}}$.

The real protocol $\mathcal{P}_{\text{iso}}$ consists of three roles, `initiator`, `responder`, and `setup`. The `setup` role models secure generation and distribution of a system parameter, namely, a description of a cyclic group (G, n, g). As this parameter must be shared between all runs of a key exchange protocol, `setup` is implemented by a single machine which spawns a single instance that manages all entities and always outputs the same parameter. The roles `initiator` and `responder` implement parties A and B, respectively, from Figure 8. Each role is implemented by a separate machine and every instance of those machines manages exactly one entity. Thus, these instances directly correspond to an actual implementation where each run of a key exchange protocol spawns a new program instance. Both, initiator and responder register their public keys at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ before they actually start the key exchange.

In $\mathcal{P}_{\text{iso}}$, we use a standard ideal signature functionality $\mathcal{F}_{\text{sig}}$[16] for signing messages and verifying signatures. As common, the ideal functionality $\mathcal{F}_{\text{sig}}$ consists of two roles, `signer` and `verifier`, that allows for the corresponding operations. Both roles are implemented by the same machine and instances of that machine manage entities that share the same SID. The SID sid of an entity is structured as a tuple $(pid_{owner}, (sid', pid_i, pid_r))$, modeling a specific key pair of the party pid_{owner}. More specifically, in protocol $\mathcal{P}_{\text{iso}}$, every party pid owns a single key pair per key exchange session, represented by SID $(pid, (sid', pid_i, pid_r))$. We use the accountable PKI functionality $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ as PKI. To simplify presentation, we use a fixed session of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ (namely $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ runs in the session with SID ε). This, however, can trivially be extended to the case where multiple PKIs are used by different key exchanges. We use an initialization functionality $\mathcal{F}_{\text{init}}^{\text{iso}}$ which ensures that initiator and responder and their judges are initialized "at the same time". This simplifies the proof later on, as we do not have to take care of several edge cases in cases of non-initialized parties.

The corruption model of $\mathcal{P}_{\text{iso}}$ allows $\mathcal{A}$ to corrupt users $(pid, sid, role)$ (statically) before the key exchange starts. We consider a party corrupted, if *(i)* a party is directly corrupted, *(ii)* the instance of the party at $\mathcal{F}_{\text{sig}}$, i. e., its "private signature key", is corrupted, *(iii)* its verification instance at $\mathcal{F}_{\text{sig}}$, used to verify the intended partners signature, is corrupted, and *(iv)* its session at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ is corrupted. This essentially models a party-wise corruption.

Additionally to $\mathcal{P}_{\text{iso}}$, $\mathcal{P}_{\text{iso}}^{\text{acc}}$ adds local judges to $\mathcal{P}_{\text{iso}}$ – one local judge per signer, resp. verifier instance of $\mathcal{P}_{\text{iso}}$ – and a dummy supervisor. For the local judges in $\mathcal{P}_{\text{iso}}^{\text{acc}}$, we essentially pass-through the accountability properties from $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. The judge queries for verdicts at its own local judge (instance) at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ and the judge of the intended partner. In case that there is a verdict from $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ from the intended partners judge, $\mathcal{P}_{\text{judge}}^{\text{iso}}$ adds the intended partner as potentially misbehaving party to the verdict (i. e., adds $\lor$dis(intended partner)), if the intended partner's judge provides a verdict from the lower level. This captures that the judge cannot be certain who behaved maliciously on the PKI level: the issuing CA or the intended partner.

[16]In contrast to Appendix E, where we use an adapted variant of $\mathcal{F}_{\text{sig}}$. Additionally, we also allow $\mathcal{A}$ to corrupt the `verifier` instance here.

Description of the protocol $\mathcal{F}_{\text{KE}}^{\text{acc}} = (\mathtt{initiator}, \mathtt{responder}, \mathtt{supervisor}, \mathtt{judge})$:

Participating roles: $\{\mathtt{initiator}, \mathtt{responder}, \mathtt{judge}, \mathtt{supervisor}\}$
Corruption model: *dynamic with secure erasures*
Protocol parameters:

- $\mathsf{groupGen}(1^\eta)$. {*Algorithm for generating tuples (G, n, g) describing cyclic groups G of size n with generator g.*
- $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$ {*Accountability properties, meant to be set to* $\{\mathtt{authenticity}\}$
- $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$ {*Assumption-based security properties, meant to be set to* $\emptyset$
- $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$ {*set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)*
- $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$ {*set of entities/IDs where properties are ensured via assumptions*

Description of $M_{\mathtt{initiator},\mathtt{responder},\mathtt{supervisor},\mathtt{judge}}$:

```
Implemented role(s): $\{\mathtt{initiator}, \mathtt{responder}, \mathtt{judge}, \mathtt{supervisor}\}$
Subroutines: $\mathcal{F}^{\text{KE}}_{\text{judgeParams}}$
Internal state:
  – $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$        {Global group parameters.
  – $\mathsf{state} : (\{0,1\}^*)^3 \to \{\bot, \mathtt{registered}, \mathtt{started}, \mathtt{finished}\}$        {Stores the current state of entities in key exchange; initially $\bot$.
  – $\mathsf{initiator} : (\{0,1\}^*)^3$        {The initiator of the key exchange
  – $\mathsf{responder} : (\{0,1\}^*)^3$        {The responder of the key exchange
  – $\mathsf{caller} : \{\mathsf{initiator}, \mathsf{responder}\} \to (\{0,1\}^*)^3 \cup \{\bot\}$        {Stores the calling entity for both entities in key exchange; initially $\bot$.
  – $\mathsf{sessionKey} : \{0,1\}^* \cup \{\bot\}$        {The session key; initially $\bot$.
  – $\mathsf{corruptedIntParties} \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\mathtt{judge}, \mathtt{supervisor}\})$, initially $\emptyset$        {The set of corrupted internal parties $(pid, sid, role)$
  – $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\mathtt{true}, \mathtt{false}\}$        {Stores broken security assumptions per id, initially $\mathtt{false}$ $\forall$entries
  – $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\mathtt{true}, \mathtt{false}\}$        {Stores broken security properties per judge/id, initially $\mathtt{false}$ $\forall$entries
  – $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$        {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
CheckID$(pid, sid, role)$:
    Accept all entities of form $(pid, (sid', pid_i, pid_r), role)$ where $role \in \{\mathtt{initiator}, \mathtt{responder}\}$
    and $pid \in \{pid_i, pid_r\}$ or $((\mathtt{local}, pid', role'), (sid', pid_i, pid_r), \mathtt{judge})$ where $role' \in$
    $\{\mathtt{initiator}, \mathtt{responder}\}$ and also $pid' \in \{pid_i, pid_r\}$ or $(\_, (sid', pid_i, pid_r), \mathtt{supervisor})$.
        {One session of $\mathcal{F}^{\text{acc}}_{\text{KE}}$ manages a single instance of a key exchange.
Corruption behavior:
  – AllowCorruption$(pid, sid, role)$:
      Do not allow corruption of $(pid, sid, \mathtt{supervisor})$.
      if $role = \mathtt{judge}$:
        send $(\mathsf{Corrupt}, (pid, sid, \mathtt{judge}), \mathsf{internalState})$
         to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \mathtt{judgeParams})$        {$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted
        wait for $b$; return $b$
  – DetermineCorrStatus$(pid, sid, role)$:
      if $role = \mathtt{judge}$:        {$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status
        send $(\mathtt{CorruptionStatus?}(pid, sid, \mathtt{judge}), \mathsf{internalState})$
         to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \mathtt{judgeParams})$
        wait for $b$; return $b$
  – AllowAdvMessage$(pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m)$        {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$.
      Do not allow sending messages to $\mathcal{F}_{\text{judgeParams}}$.
  – LeakedData$(pid, sid, role)$:
    If called while $(pid, sid, role)$ determines its initial corruption status, use the default behavior of LeakedData.
    That is, output the initially received message and the sender of that message.
    Otherwise, return $(\mathsf{caller}[pid, sid, role], \mathsf{sessionKey})$.
Initialization:
    Parse $\mathsf{sid}_{\mathsf{cur}}$ as $(sid', pid_i, pid_r)$.
    $\mathsf{initiator} \leftarrow (pid_i, (sid', pid_i, pid_r), \mathtt{initiator})$        {Store initiator and responder
    $\mathsf{responder} \leftarrow (pid_r, (sid', pid_i, pid_r), \mathtt{responder})$
    $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow \mathsf{groupGen}(1^\eta)$.
    if $sender = \mathtt{NET} \land m = \mathtt{InitGroup}$:
      send $(\mathtt{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ to $\mathtt{NET}$.        {Allow adversary to start initialization and then return the generated group. No other actions are performed in this case.
    else:
      send responsively $(\mathtt{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ to $\mathtt{NET}$;        {Leak group parameters to the adversary. Note that this command forces the adversary to respond before interacting with the protocol in any other way, i.e., the run can continue as expected.
      wait for $\_$.
MessagePreprocessing:
    if $\mathsf{role}_{\mathsf{cur}} = \mathtt{initiator} \land \mathsf{caller}[\mathsf{initiator}] = \bot$:
      $\mathsf{caller}[\mathsf{initiator}] \leftarrow \mathsf{entity}_{\mathsf{call}}$
    else if $\mathsf{role}_{\mathsf{cur}} = \mathtt{responder} \land \mathsf{caller}[\mathsf{responder}] = \bot$:
      $\mathsf{caller}[\mathsf{responder}] \leftarrow \mathsf{entity}_{\mathsf{call}}$
Main:
    recv $\mathtt{RegisterPubKey}$ from $\mathtt{I/O}$ to $(\_, \_, \mathtt{initiator}) \lor (\_, \_, \mathtt{responder})$ s.t. $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] = \bot$:        {Register that initiator and responder want exchange keys
      $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] \leftarrow \mathtt{registered}$.
      send $\mathtt{RegisterPubKey}$ to $\mathtt{NET}$.
    recv $\mathtt{InitKE}$ from $\mathtt{I/O}$ to $(\_, \_, \mathtt{initiator}) \lor (\_, \_, \mathtt{responder})$ s.t. $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] = \mathtt{registered}$:        {Start key exchange key
      $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] \leftarrow \mathtt{started}$.
      send $\mathtt{InitKE}$ to $\mathtt{NET}$.
```

[a]Here: $\mathsf{Roles}_{\mathcal{F}} := \{\mathtt{initiator}, \mathtt{responder}\}$.

Fig. 32: The accountable ideal key exchange functionality $\mathcal{F}^{\text{acc}}_{\text{KE}}$ (part 1).

Description of $M_{\mathsf{initiator},\mathsf{responder},\mathtt{supervisor},\mathtt{judge}}$ (continued):

```
Main:                                       {Note that Main continues processing the message m that Initialization has
                                             already parsed if Initialization does not end the run.
        recv $(\mathtt{FinishKE}, k)$ from NET to $(\_, \_, \mathtt{initiator}) \lor (\_, \_, \mathtt{responder})$ s.t. $\mathsf{state}[(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})] = \mathtt{started}$:
            if $\mathsf{sessionKey} = \bot$:
                Choose $\mathsf{sessionKey} \leftarrow \mathsf{G}$ uniformly at random
            $\mathsf{state}[\mathsf{entity_{cur}}] \leftarrow \mathtt{finished}$.
            if $\mathsf{initiator} \in \mathsf{CorruptionSet} \lor \mathsf{responder} \in \mathsf{CorruptionSet} \lor$
              $\mathsf{brokenProps}[\mathtt{authenticity}, (\mathtt{local}, \mathtt{initiator}, \mathtt{initiator})] = \mathtt{true} \lor$
              $\mathsf{brokenProps}[\mathtt{authenticity}, (\mathtt{local}, \mathtt{responder}, \mathtt{responder})] = \mathtt{true}$:
                send $(\mathtt{FinishKE}, k)$ to $\mathsf{caller}[\mathsf{entity_{cur}}]$.        {$\mathcal{A}$ may choose the key if he has corrupted the partner or
                                                                      authenticity is broken for one of the involved parties.
            else:
                send $(\mathtt{FinishKE}, \mathsf{sessionKey})$ to $\mathsf{caller}[\mathsf{entity_{cur}}]$.

      Include static code from the AUC transformation $\mathcal{T}_1(\cdot)$ here, i.e., include additional code from Figure 2 and 3 here. Do not include the public
      judge here.
```

Fig. 33: The accountable ideal key exchange functionality $\mathcal{F}_{\mathrm{KE}}^{\mathrm{acc}}$ (part 2).

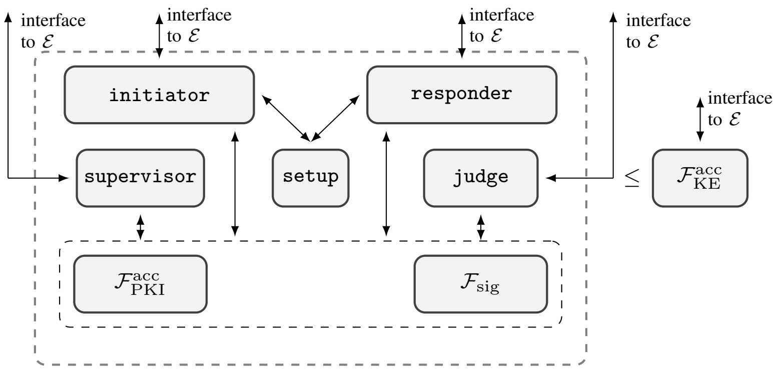
Description of $\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}} = (\mathtt{judgeParams})$:

Participating roles: $\{\mathtt{judgeParams}\}$
Corruption model: *incorruptible*

Description of $M_{\mathtt{judgeParams}}$:

```
Implemented role(s): $\{\mathtt{judgeParams}\}$
CheckID$(pid, sid, role)$:
    Accept all messages with the same $sid$.
Main:
    recv $(\mathtt{BreakAccProp}, verdict, toBreak, internalState)$ from I/O:
      Let initiator, responder be initiator and responder extracted from $internalState$.Further, let $pid_{\mathsf{initiator}}$, resp. $pid_{\mathsf{responder}}$ be their PIDs.
        if $toBreak = \{\mathtt{authenticity}\} \times \underbrace{(\{\mathtt{local}\} \times \{(pid_{\mathsf{initiator}}, \mathtt{initiator}), (pid_{\mathsf{responder}}, \mathtt{responder})\})}_{=: ids}$:
                                                              {$s$ is defined to be the session ID of the considered session
            for all $id \in ids$ do:
                if $verdict$ for $id$ does not match conjunction of verdicts of form
                  $\mathsf{dis}((pid', sid', role')) \lor V$ or $V$
                  where $(pid', sid', role') \in \{\mathsf{initiator}, \mathsf{responder}\}$ and
                  $V$ is a verdict not blaming parties with roles from $\{\mathtt{initiator}, \mathtt{responder}, \mathtt{judge}, \mathtt{supervisor}\}$:
            Let $k$ be $\mathsf{sessionKey}$ extracted from $\mathcal{F}_{\mathrm{KE}}^{\mathrm{acc}}$'s provided internal state.
            reply $(\mathtt{BreakAccProp}, \mathtt{true}, k)$                  {Accept verdict/broken properties, leak session key
        else:
            reply $(\mathtt{BreakAccProp}, \mathtt{false}, \varepsilon)$                      {Decline verdict/broken properties
    recv $(\mathtt{GetJudicialReport}, msg, internalState)$ from I/O:                    {Generate judicial report
        reply $(\mathtt{GetJudicialReport}, \varepsilon)$                          {We do not provide a judicial report here
    recv $(\mathtt{BreakAssumption}, toBreak, internalState)$ from I/O:          {Do not generate leakage when breaking assumptions
        reply $(\mathtt{BreakAssumption}, \varepsilon)$
    recv $(\mathtt{Corrupt}, (id, sid, \mathtt{judge}), internalState)$ from I/O:      {$\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$ allows to corrupt a local judge iff the
        if $id = (\mathtt{local}, pid, role) \land (pid, sid, role) \in \mathsf{CorruptionSet}$:   accompanied initiator or responder is corrupted
            reply true
        else:
            reply false
    recv $(\mathtt{CorruptionStatus?}, (id, sid, \mathtt{judge}), internalState)$ from I/O:   {$\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$ interprets a local judge as corrupted iff
        if $(id, sid, \mathtt{judge}) \in \mathsf{CorruptionSet}$:                       the accompanied initiator or responder is corrupted
            reply true
        else if $id = (\mathtt{local}, pid, role) \land (pid, sid, role) \in \mathsf{CorruptionSet}$ :
            reply true
        else:
            reply false
```

Fig. 34: The judge parameter functionality $\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$ for $\mathcal{F}_{\mathrm{KE}}^{\mathrm{acc}}$.



interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
interface to $\mathcal{E}$
initiator
responder
supervisor
setup
judge
$\leq$
$\mathcal{F}^{\text{acc}}_{\text{KE}}$
$\mathcal{F}^{\text{acc}}_{\text{PKI}}$
$\mathcal{F}_{\text{sig}}$

Fig. 35: Realization relation of a key exchange example stated in Theorem 8. The system $\mathcal{E}$ denotes the environment, modeling, as usual in UC setting, arbitrary higher level protocols. All machines are additionally connected to the network adversary.

As we do not consider assumption-based security properties in this case study, the supervisor $\mathcal{P}^{\text{iso}}_{\text{sv}}$ always responds with `false` if a party queries for the status of an assumption. Further, $\mathcal{P}^{\text{iso}}_{\text{sv}}$ allows querying for the corruption status of (internal) parties in $\mathcal{F}^{\text{acc}}_{\text{PKI}}$.

F.3 Security Analysis: We now provide the formal result that we can use the ISO protocol on top of an accountable PKI to realize the (accountable) key exchange functionality $\mathcal{F}^{\text{acc}}_{\text{KE}}$.

Theorem 8. *Let* $\mathsf{groupGen}(1^\eta)$ *be an algorithm that outputs descriptions* (G, n, g) *of cyclic groups (i. e., G is a group of size n with generator g) such that n grows exponentially in η and the DDH assumption holds true. Let* $\mathcal{P}^{\text{acc}}_{iso}$ *be the (accountable) ISO protocol as described above, let* $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ *be the accountable ideal PKI functionality and* $\mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}$ *the associated subroutine both with parameters* $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{correctCert}\}$ *and* $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$*, and let* $\mathcal{F}_{\text{sig}}$ *be the (standard) ideal signature functionality, and let* $\mathcal{F}^{\text{acc}}_{\text{KE}}$ *be the (accountable) ideal functionality for authenticated key exchange with* $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{authenticity}\}$, $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$, $\mathsf{pids}_{\mathsf{judge}} = \{\mathtt{local}\} \times (\{0,1\}^*)^2$*, and* $\mathsf{ids}_{\mathsf{assumption}} = \emptyset$*. Then the following holds true:*

$$(\mathcal{P}^{\text{acc}}_{iso}, \mathcal{P}^{iso}_{\text{sv}}, \mathcal{P}^{iso}_{\text{judge}} \mid \mathcal{F}^{\text{acc}}_{\text{PKI}}, \mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}, \mathcal{F}_{\text{sig}}) \leq$$
$$(\mathcal{F}^{\text{acc}}_{\text{KE}} \mid \mathcal{F}^{\text{KE}}_{\text{judgeParams}}).$$

Proof. We show that the real key exchange $\mathcal{R} := (\mathcal{P}^{\text{acc}}_{\text{iso}}, \mathcal{P}^{\text{iso}}_{\text{sv}}, \mathcal{P}^{\text{iso}}_{\text{judge}} \mid \mathcal{F}^{\text{acc}}_{\text{PKI}}, \mathcal{F}^{\text{acc-PKI}}_{\text{judgeParams}}, \mathcal{F}_{\text{sig}})$ realizes the ideal key exchange $\mathcal{I} := (\mathcal{F}^{\text{acc}}_{\text{KE}} \mid \mathcal{F}^{\text{KE}}_{\text{judgeParams}})$. As part of this, we define a responsive simulator $\mathcal{S}$ such that the real world running $\mathcal{R}$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$ for every ppt environment $\mathcal{E}$.

First note that it is easy to see that both $\mathcal{R}$ and $\mathcal{I}$ are environmentally bounded and complete. Now, the simulator $\mathcal{S}$ is defined as follows: the simulator is a single machine that is connected to $\mathcal{I}$ and the environment via their network interfaces. In a run, there is only a single instance of the machine $\mathcal{S}$ that accepts all incoming messages. The simulator $\mathcal{S}$ internally simulates the *full protocol* $\mathcal{R}$, including its behavior on the network interface to the environment. To improve readability, we refer to $\mathcal{S}$'s simulated version of the real protocol as $\mathcal{R}'$. More precisely, the simulation runs as follows:

- At the start of a run, $\mathcal{S}$ obtains the group parameters used by $\mathcal{F}^{\text{acc}}_{\text{KE}}$: if the simulator is activated for the first time via the $(\mathsf{LeakGroup}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$ message, then he simply saves this message and returns `ok`. Otherwise, $\mathcal{S}$ sends an `InitGroup` message to (an arbitrary entity of) $\mathcal{F}^{\text{acc}}_{\text{KE}}$ to trigger **Initialization** and obtain the group parameters. Note that the environment cannot observe whether the simulator has manually triggered **Initialization** of $\mathcal{F}^{\text{acc}}_{\text{KE}}$. The group parameters are used by $\mathcal{S}$ as output of the internally simulated $\mathcal{P}^{\text{acc}}_{\text{iso}}$: `setup` role.
- Upon receiving `RegisterPubKey` from an honest party,[17] $\mathcal{S}$ forwards this message to the dedicated machine of $\mathcal{R}'$. We note that $\mathcal{S}$ can derive all involved parties and their roles from the SID (which predefines initiator

[17]We consider an entity to be honest if it outputs `false` upon `CorruptionStatus`? requests. Conversely, we call an entity corrupted if it outputs `true`, even if it was not explicitly corrupted.

and responder of the key exchange session including their local judges). In $\mathcal{R}'$, the message then triggers the initialization of the party's signer instance at $\mathcal{F}_{\text{sig}}$ and registers the generated public key for the dedicated party at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$.

- Upon receiving a `InitKE` from an honest entity, $\mathcal{S}$ forwards this message – again – in the name of a higher-level protocol to the simulated entity in $\mathcal{R}'$. This triggers the start of a key exchange as defined in the ISO protocol in $\mathcal{R}'$.
- As soon as an honest `initiator` in $\mathcal{R}'$ outputs $(\texttt{FinishKE}, \mathsf{sessionKey})$, $\mathcal{S}$ sends the message $(\texttt{FinishKE}, \mathsf{sessionKey})$ to the initiator identity in $\mathcal{F}_{\text{KE}}^{\text{acc}}$.
- As soon as an honest `responder` in $\mathcal{R}'$ outputs $(\texttt{FinishKE}, \mathsf{sessionKey})$, $\mathcal{S}$ distinguishes three cases:
  1. In the case that `initiator` is corrupted after its output of `FinishKE`, $\mathcal{I}$ leaks the initiator's ideal key k' to $\mathcal{S}$. Then, $\mathcal{S}$ sends $(\texttt{FinishKE}, k')$ to $\mathcal{I}$.
  2. If `authenticity` is broken for (at least) one of both parties after `initiator` outputs the session key via `FinishKE`. In this case, $\mathcal{S}$ learns the (ideal) session key k' from $\mathcal{F}_{\text{KE}}^{\text{acc}}$ when providing the verdict to $\mathcal{F}_{\text{KE}}^{\text{acc}}$ before outputting `FinishKE`. Then, $\mathcal{S}$ sends $(\texttt{FinishKE}, k')$ to $\mathcal{I}$.
  3. In all other cases, $\mathcal{S}$ sends the message $(\texttt{FinishKE}, \mathsf{sessionKey})$ to the responder identity in $\mathcal{F}_{\text{KE}}^{\text{acc}}$.
- $\mathcal{S}$ forwards all network communication from the environment to corresponding entities in $\mathcal{R}'$, and vice versa.
- $\mathcal{S}$ keeps the (internal) corruption status of entities in roles of $\mathcal{R}'$ and $\mathcal{I}$ synchronized. In particular, if an entity of $\mathcal{I}$ asks for its initial corruption status, then the same entity in $\mathcal{R}'$ is simulated to also do so. Furthermore, as soon as a simulated entity of $\mathcal{R}'$ considers itself to be corrupted (either explicitly or due to a corrupted subroutine), $\mathcal{S}$ corrupts the same entity in $\mathcal{I}$.
- If a corrupted entity *entity* in a public role of $\mathcal{R}$ outputs a message on its I/O interface to a higher-level protocol, then $\mathcal{S}$ instructs the (explicitly) corrupted entity *entity* in $\mathcal{I}$ to forward the same message on its I/O interface. The same is also done in the other direction. Note that this also includes corrupted local judges.
- Always when an instance $((\texttt{local}, pid, role), sid, \texttt{judge})$ of $\mathcal{P}_{\text{judge}}^{\text{iso}}$ in $\mathcal{R}'$ renders a verdict, $\mathcal{S}$ extracts this verdict and sends $(\texttt{BreakAccProp}, verdict, toBreak)$ where $verdict$ maps $(\texttt{local}, pid, role)$ to the verdict extracted from the simulation (and all other entries to ε) and $toBreak = \{(\texttt{authenticity}, (\texttt{local}, pid, role)\}\}$ to $((\texttt{local}, pid, role), sid, \texttt{judge})$ in $\mathcal{I}$.
- As the supervisor does not have to handle assumption-based properties, there is no additional specification for $\mathcal{S}$ necessary.

This concludes the description of the simulator. It is easy to see that $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded and $\mathcal{S}$ is responsive for $\mathcal{I}$. We note that the behavior of the supervisor is indistinguishable between $\mathcal{R}$ and $\mathcal{I}$ as he always provides the same output in both worlds. Now, let $\mathcal{E}$ be an arbitrary but fixed responsive environment.

We will now go over all possible interactions with honest entities on the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the environment, i.e., $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable. At the start of a run, there were no interactions on the I/O interface with honest parties yet, i.e., $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable. In the following, the induction assumption is that all I/O interactions to far have resulted in the same behavior visible towards the environment in both the real and ideal world.

We firstly have a look at the I/O output of `initiator` and `responder`. We then prove that the output of local judges is also indistinguishable between the $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$.

Initiator and responder. We note that all I/O calls to honest `initiator` and `responder` do not directly lead to an output to $\mathcal{E}$ in $\mathcal{I}$ and $\mathcal{R}'$ (for the corrupted case, see below). Thus, $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable. However, note that the phases of the protocol are synchronized in $\mathcal{I}$ and $\mathcal{R}'$. `RegisterPubKey` initializes the signatures key at $\mathcal{F}_{\text{sig}}$ and also registers the public key at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ in $\mathcal{R}'$ which does not have an adequate in $\mathcal{I}$. `InitKE` starts the key exchange itself in both worlds.

Note that honest `initiator` or `responder` do not send `FinishKE` to $\mathcal{E}$ if their partners public key is not registered at the PKI $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ or if the signatures provided via NET do not verify. Also note that a party initialization leads to an initialization of both parties, initiator and responder and also their local judges. This enforces that the corruption status of the machines is always correctly handled and that we do not have to take care of edge cases due to uninitialized parties.

When the `initiator` in $\mathcal{R}'$ wants to output the message $(\texttt{FinishKE}, \mathsf{sessionKey})$ to $\mathcal{E}$, $\mathcal{S}$ triggers $\mathcal{I}$ via $(\texttt{FinishKE}, \mathsf{sessionKey})$ also to output a session key. If initiator and responder are uncorrupted and the property `authenticity` holds true for both, $\mathcal{I}$ outputs a randomly selected group element (potentially $\neq$ $\mathsf{sessionKey}$) from $\mathsf{h}_{\mathsf{init}}$ on behalf of `initiator`. As $\mathcal{E}$ is not aware of the $\mathsf{sessionKey}$ (both parties are honest) and as the DDH assumption holds true, $\mathcal{E}$ can only distinguish with negligible probability between $\mathcal{I}$ and $\mathcal{R}'$.

If both parties are still uncorrupted but `authenticity` is broken for at least one of both parties, this leads to the case that $\mathcal{I}$ directly outputs the $\mathsf{sessionKey}$ provided by $\mathcal{S}$. As the session key is extracted from $\mathcal{R}'$, this case is also indistinguishable between $\mathcal{R}$ and $\mathcal{I}$.

In case that `initiator` considers itself as corrupted in $\mathcal{R}'$, `responder` outputs the $\mathsf{sessionKey}$. Note that this session key matches the one exchanged with `initiator` thus potentially leaked to $\mathcal{A}$. Thus, $\mathcal{E}$ cannot distinguish between $\mathcal{I}$ and $\mathcal{R}$.

The same holds true for the case, `responder` is corrupted but `initiator` is not.

The two cases above also include the cases when `authenticity` is additionally broken for one party – as this does not change the behavior of $\mathcal{I}$ in the case that already one of the parties is corrupted.

Note that we have to consider two special cases here:

1. `initiator` gets corrupted after it outputted the key. However, as $\mathcal{F}^{\mathrm{acc}}_{\mathrm{KE}}$ leaks the key upon corruption to $\mathcal{S}$ and $\mathcal{S}$ forwards this leaked key to the `responder`, $\mathcal{E}$ cannot distinguish between $\mathcal{I}$ and $\mathcal{R}$.
2. `initiator` or `responder` are both honest but one of them/both lose/s `authenticity` after the `initiator` already outputted `FinishKE` including an ideally generated session key to $\mathcal{E}$. However, as $\mathcal{F}^{\mathrm{acc}}_{\mathrm{KE}}$ leaks the key upon breaking to $\mathcal{S}$ and $\mathcal{S}$ forwards this leaked key to the `responder`, $\mathcal{E}$ cannot distinguish between $\mathcal{I}$ and $\mathcal{R}$.

Judges. Note that $\mathcal{S}$ updates verdicts in $\mathcal{I}$ as soon as they occur in $\mathcal{R}'$. Further note that `GetVerdict` and `GetJudicialReport` do not influence any state in $\mathcal{R}$ and $\mathcal{I}$. We also note that answers to `GetJudicialReport` are indistinguishable in both worlds.

Further, $\mathcal{P}^{\mathrm{iso}}_{\mathrm{judge}}$ always accepts the input of $\mathcal{S}$ as rendered verdicts in $\mathcal{R}'$ are compliant with the restrictions imposed in $\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$. Thus, local judges output the same verdicts in $\mathcal{I}$ and $\mathcal{R}'$ and we can conclude that both worlds are indistinguishable.

As local judges act as pure message forwarder when their associated party is corrupted, we can also conclude that $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable in this case.

Supervisor. Supervisors is always the same in $\mathcal{R}$ and $\mathcal{I}$. Thus, both worlds are indistinguishable regarding their output.

We can also conclude that the corruption status of (internal) entities in the real and ideal world is synchronized. Since the simulator has full control over corrupted entities, which are handled via the internal simulation $\mathcal{R}'$, this implies that the I/O behavior of corrupted entities of $\mathcal{R}/\mathcal{I}$ towards the environment is also identical in the real and ideal world.

Note that the behavior of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ is only visible to $\mathcal{I}$ via its network interface. As $\mathcal{S}$ forwards these messages to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ in $\mathcal{R}'$ and also provide the output of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ to $\mathcal{E}$ via NET, $\mathcal{E}$ cannot use $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ to distinguish between $\mathcal{I}$ and $\mathcal{R}$.

Overall, we can conclude that $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable.

□

F.4 Deterrence Analysis: Please note that we consider the fully composed protocol here, i. e., including the used PKI. We emphasize that the meaning of a single established key exchange is negligible in contrast to a working PKI. Therefore, being caught cheating in the PKI is the major deterrence to behave honestly in the key exchange. In particular, we conclude that the utilities for PKI participants have the same relations as in Appendix E.4, i. e., Equation 1 and 2 are met for these parties. For the parties involved in the key exchange, we assume that $U^i_{hP} = U^i_{hE} = U^i_{hL} = U^i_{mP} = U^i_{mL} = U^i_{mE} = 0$ expressing that a party typically does not have a big advantage from executing the key exchange honest or maliciously. Thus, Equation 1 and 2 holds true for all involved parties.

G. Capturing MPC Accountability Properties via AUC

As already mentioned in the introduction, there are several works that capture accountability properties in a UC model for the special case of MPC protocols (e.g., [13, 14, 24, 34, 35, 55, 74, 81]). These properties include *(publicly) identifiable abort* [14, 34, 55]), *(public/universal) verifiability* [35, 84, 86], *auditability* [13], Description of the protocol $\mathcal{P}^{\text{acc}}_{\text{iso}} = (\texttt{initiator}, \texttt{responder} \mid \texttt{setup})$:

Participating roles: initiator, responder, setup
Corruption model: static
Protocol parameters:
- $\mathsf{groupGen}(1^\eta)$. $\{$*Algorithm for generating tuples (G, n, g) describing cyclic groups G of size n with generator g.*

Description of $M_{\texttt{initiator}}$:

Implemented role(s): initiator
Subroutines: setup, $\mathcal{F}_{\text{sig}}$, $\mathcal{F}^{\text{acc}}_{\text{PKI}}, \mathcal{F}^{\text{iso}}_{\text{init}}$: init
Internal state:
- $\mathsf{pk} \in \{0,1\}^*, \mathsf{pk} = \varepsilon, \mathsf{pid}_{\mathtt{CA}} \in \{0,1\}^*, \mathsf{pid}_{\mathtt{CA}} = \varepsilon$ $\{$*The public key of* $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ *and the used CA*
- $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\bot\})^3 = (\bot, \bot, \bot)$ $\{$*Global group parameters.*
- $\mathsf{state} \in \{\bot, \texttt{registered}, \texttt{started}, \texttt{finished}\} = \bot$ $\{$*Current state in key exchange.*
- $\mathsf{initiator} : (\{0,1\}^*)^3$ $\{$*The initiator of the key exchange*
- $\mathsf{responder} : (\{0,1\}^*)^3$ $\{$*The responder of the key exchange*
- $\mathsf{caller} \in (\{0,1\}^*)^3 \cup \{\bot\} = \bot$ $\{$*Stores the initial caller of this entity/instance.*
- $\mathsf{sessionKey} \in \{0,1\}^* \cup \{\bot\} = \bot$ $\{$*Stores the session key.*
- $\mathsf{e}_{\mathsf{init}} \in \mathbb{Z}_n \cup \{\bot\} = \bot$ $\{$*Secret DH exponent of initiator.*
- $\mathsf{h}_{\mathsf{resp}} \in \mathsf{G} \cup \{\bot\} = \bot$ $\{$*Public DH key share of responder.*

CheckID$(pid, sid, role)$**:**
Accept all entities of form $(pid_i, (sid', pid_i, pid_r), \texttt{initiator})$.

Corruption behavior:
- **DetermineCorrStatus**$(pid, sid, role)$: $\{$*Entity corrupted if one of its signature keys or the verification subroutine are corrupted.*

```
    $out \leftarrow \mathbf{corr}(pid, (pid, sid), \mathcal{F}_{\text{sig}} : \texttt{signer}) \lor \mathbf{corr}(pid, (pid(\mathsf{responder}), sid), \mathcal{F}_{\text{sig}} : \texttt{verifier})$    $\{$ $pid(\mathsf{responder})$ denotes $\mathsf{responder}$'s PID
    $out \leftarrow out \lor \mathbf{corr}((pid, sid, role), \varepsilon, \mathcal{F}^{\text{acc}}_{\text{PKI}} : \texttt{client})$
    return $out$
```

- **AllowAdvMessage**$(pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m)$:
  If $role_{receiver} = \mathcal{F}_{\text{sig}}$: signer, return false.[a]
  Otherwise output true iff $pid = pid_{\text{receiver}}$.

Initialization:

```
Parse $\mathsf{sid}_{\mathsf{cur}}$ as $(sid', pid_i, pid_r)$.
$\mathsf{initiator} \leftarrow (pid_i, (sid', pid_i, pid_r), \texttt{initiator})$.    $\{$Store initiator and responder.
$\mathsf{responder} \leftarrow (pid_r, (sid', pid_i, pid_r), \texttt{responder})$.
$\mathsf{caller} \leftarrow (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$.
send GetParameters to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \texttt{setup})$;    $\{$Get DH parameters
wait for $(\texttt{GetParameters}, (G, n, g))$.
$(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow (G, n, g)$.
if ITM was not activated via init macro, resp. a InitEntity message:
    send initPartner to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\text{iso}}_{\text{init}} : \texttt{init})$
    wait for initPartner
```

Main:
See Figure 37.

[a]In our modeling, the corruption status of signer entities indicates whether the adversary has access to the corresponding signature keys, i.e., whether he can sign his own messages (as in this case the signer entity should be considered compromised). Thus, the adversary is not allowed to access uncorrupted signer entities. If the signer entity is corrupted, then the adversary already knows the secret key and can sign messages on his own, so there is no need to give him access in this case.

Fig. 36: A real key exchange protocol $\mathcal{P}^{\text{acc}}_{\text{iso}}$ for realizing $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ (part 1). Note that each instance of $M_{\texttt{initiator}}$ and $M_{\texttt{responder}}$ corresponds to a single entity.

Description of $M_{\mathtt{initiator}}$ (continued):

```
Main:
    recv RegisterPubKey from I/O s.t. $\mathsf{state}[(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})] = \perp$:    {Register initiator's signature and certificate
        $\mathsf{state}[(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}})] \leftarrow \mathtt{registered}$.
        send InitSign to $(\mathsf{pid_{cur}}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{sig}} : \mathtt{signer})$;
        wait for $(\mathtt{InitSign}, \mathtt{success}, pk)$; $\mathsf{pk} \leftarrow pk$;
        send responsively $(\mathtt{Register}, \mathsf{pk})$ to NET;
        wait for $(\mathtt{Register}, \mathsf{pk}, pid_{\mathtt{CA}})$ s.t. $pid_{\mathtt{CA}} \in \{0,1\}^*$;
        $\mathsf{pid}_{\mathtt{CA}} \leftarrow pid_{\mathtt{CA}}$;
        send $(\mathtt{Register}, \mathsf{pk}, pid_{\mathtt{CA}})$ to $((\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), \varepsilon, \mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}} : \mathtt{client})$.[a]    {Register identity at $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$
    recv InitKE from I/O s.t. $\mathsf{state} = \mathtt{registered}$:    {Start KE and send first message.
        $\mathsf{state} \leftarrow \mathtt{started}$.
        Choose $\mathsf{e_{init}} \leftarrow \mathbb{Z}_n$ uniformly at random, compute $h_{init} = \mathsf{g}^{\mathsf{e_{init}}}$.
        send $(\mathtt{Send}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), h_{init})$ to NET.
    recv $(\mathtt{FinishKE}, \mathsf{responder}, h_{resp}, (\mathsf{g}^{\mathsf{e_{init}}}, h_{resp}, \mathsf{responder}), \sigma)$ from NET s.t. $\mathsf{state} = \mathtt{started}$:    {Receive second message and output key.
        send responsively $(\mathtt{RetrieveCA}, \mathsf{responder})$ to NET    {Query $\mathcal{A}$ for responder's CA;
        wait for $(\mathtt{RetrieveCA}, \mathsf{responder}, pid_{\mathtt{CA}})$;
        send $(\mathtt{Retrieve}, \mathsf{responder}, pid_{\mathtt{CA}})$ to $((\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), \varepsilon, \mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}} : \mathtt{client})$;
        wait for $(\mathtt{Retrieve}, \mathsf{intendedPartner}, pid_{\mathtt{CA}}, pk)$.    {Get public verification key of intended partner.
        if $pk = \perp$:
            abort.
        send $(\mathtt{Verify}, (\mathsf{g}^{\mathsf{e_{init}}}, h_{resp}, \mathsf{responder}), \sigma, pk)$ to $(\mathsf{pid_{cur}}, (pid(\mathsf{responder}), \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{sig}} : \mathtt{verifier})$;    {$pid(\mathsf{responder})$ denoted the PID of responder
        wait for $(\mathtt{VerResult}, b)$.
        if $b = \mathtt{false}$:
            abort.
        $\mathsf{h_{resp}} \leftarrow h_{resp}$; $\mathsf{sessionKey} \leftarrow \mathsf{h_{resp}}^{\mathsf{e_{init}}}$; $\mathsf{state} \leftarrow \mathtt{finished}$.
        send $(\mathtt{FinishKE}, \mathsf{sessionKey})$ to caller.
    recv GetLastMessage from NET s.t. $\mathsf{state} = \mathtt{finished}$:
        $m = (\mathsf{h_{resp}}, \mathsf{g}^{\mathsf{e_{init}}}, \mathsf{responder})$.
        send $(\mathtt{Sign}, m)$ to $(\mathsf{pid_{cur}}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}), \mathcal{F}_{\mathrm{sig}} : \mathtt{signer})$;
        wait for $(\mathtt{Signature}, \sigma)$.
        send (send , $\sigma$) to NET.
```

[a]We consider here one instance of the PKI, namely the one with SID ε.

Description of $M_{\mathtt{setup}}$:

```
Implemented role(s): setup
Internal state:
    – $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\perp\})^3 = (\perp, \perp, \perp)$    {Global group parameters.
CheckID($pid, sid, role$): Accept all entities.
Corruption behavior:
    – AllowCorruption($pid, sid, role$) : return false.    {The adversary may not corrupt the (honestly generated) setup parameters.
Initialization:
        $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow \mathsf{groupGen}(1^\eta)$.
Main:
        recv GetParameters from _ :    {Everyone may retrieve the group parameters, including the adversary on the network.
            reply $(\mathtt{GetParameters}, (\mathsf{G}, \mathsf{n}, \mathsf{g}))$.
```

Fig. 37: A real key exchange protocol $\mathcal{P}^{\mathrm{acc}}_{\mathrm{iso}}$ for realizing $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$ (part 2).

openability [34], and *privacy* [8]. AUC can capture these accountability properties as special cases. This is not only an important sanity check but also shows that AUC generalizes and unifies existing UC accountability literature. Here we illustrate this for the most common property: identifiable abort. The other properties can be dealt with analogously, see Appendix H for more details.

Identifiable abort. The standard definition of a basic ideal MPC functionality $\mathcal{F}_{\mathrm{MPC}}$ (cf., e.g., [6, 8, 21, 24]) is based on three phases. In the first phase, it takes inputs from m parties, with inputs of corrupted parties being chosen by the adversary. In the second phase, it acts as a trusted third party that computes some function f on those inputs. In the final phase, each party receives an output of f but otherwise obtains no information. Hence, $\mathcal{F}_{\mathrm{MPC}}$ provides preventive security for correctness of the outputs and for privacy/secrecy of the inputs.

Instead of preventive security for correctness, MPC protocols often rather consider the weaker property of *identifiable abort* [6, 8, 14, 55, 85], which states that either all honest parties obtain a correct output or all honest parties agree on the name of a malicious party who has caused the output to be incorrect and hence the protocol to abort. In other words, identifiable abort is a type of (local) accountability w.r.t. correctness that additionally requires individual accountability and certain relationships between local properties of different parties.

In the literature, *identifiable abort* has been formalized within ideal functionalities $\mathcal{F}^{\text{id-ab}}_{\mathrm{MPC}}$ by letting the simulator, during the final output phase, decide whether all honest parties obtain their correct output or provide

Description of $M_{\texttt{responder}}$:

```
Implemented role(s): responder
Subroutines: setup, $\mathcal{F}_{\text{sig}}$, $\mathcal{F}_{\text{PKI}}^{\text{acc}}$, $\mathcal{F}_{\text{init}}^{\text{iso}}$ : init
Internal state:
    – $\mathsf{pk} \in \{0,1\}^*, \mathsf{pk} = \varepsilon, \mathsf{pid}_{\mathtt{CA}} \in \{0,1\}^*, \mathsf{pid}_{\mathtt{CA}} = \varepsilon$    {The public key of $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ and the used CA
    – $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \in (\{0,1\}^* \cup \{\perp\})^3 = (\perp, \perp, \perp)$    {Global group parameters.
    – $\mathsf{state} \in \{\perp, \mathtt{registered}, \mathtt{started}, \mathtt{inSession}, \mathtt{finished}\} = \perp$    {Current state in key exchange.
    – $\mathsf{initiator} : (\{0,1\}^*)^3$    {The initiator of the key exchange
    – $\mathsf{responder} : (\{0,1\}^*)^3$    {The responder of the key exchange
    – $\mathsf{caller} \in (\{0,1\}^*)^3 \cup \{\perp\} = \perp$    {Stores the initial caller of this entity/instance.
    – $\mathsf{sessionKey} \in \{0,1\}^* \cup \{\perp\} = \perp$    {Stores the session key.
    – $\mathsf{e}_{\mathsf{resp}} \in \mathbb{Z}_n \cup \{\perp\} = \perp$    {Secret DH exponent of responder.
    – $\mathsf{h}_{\mathsf{init}} \in \mathsf{G} \cup \{\perp\} = \perp$    {Public DH key share of initiator.
CheckID($pid, sid, role$):
    Accept all entities of form $(pid_r, (sid', pid_i, pid_r), \mathtt{responder})$.
Corruption behavior:
    – DetermineCorrStatus($pid, sid, role$) :
            {Consider entity corrupted if one of the signature keys or the verification subroutine is corrupted.
        $out \leftarrow \mathbf{corr}(pid, (pid, sid), \mathcal{F}_{\text{sig}} : \mathtt{signer}) \lor \mathbf{corr}(pid, (pid(\mathsf{initiator}), sid), \mathcal{F}_{\text{sig}} : \mathtt{verifier})$
        $out \leftarrow out \lor \mathbf{corr}((pid, sid, role), \varepsilon, \mathcal{F}_{\text{PKI}}^{\text{acc}} : \mathtt{client})$
        return $out$
    – AllowAdvMessage($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{receiver}, m$):
        If $role_{receiver} = \mathcal{F}_{\text{sig}} : \mathtt{signer}$, return false.    {cf. explanation in Figure 36.
        Otherwise output true iff $pid = pid_{\text{receiver}}$.
Initialization:
    Parse $\mathsf{sid}_{\mathsf{cur}}$ as $(sid', pid_i, pid_r)$.
    $\mathsf{initiator} \leftarrow (pid_i, (sid', pid_i, pid_r), \mathtt{initiator})$.    {Store initiator and responder.
    $\mathsf{responder} \leftarrow (pid_r, (sid', pid_i, pid_r), \mathtt{responder})$.
    $\mathsf{caller} \leftarrow (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$.
    send GetParameters to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{setup})$;    {Get DH parameters
    wait for $(\mathtt{GetParameters}, (G, n, g))$.
    $(\mathsf{G}, \mathsf{n}, \mathsf{g}) \leftarrow (G, n, g)$.
    if ITM was not activated via init macro, resp. a InitEntity message:
        send initPartner to $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init}}^{\text{iso}} : \mathtt{init})$
        wait for initPartner
Main:
    recv RegisterPubKey from I/O s.t. $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] = \perp$:    {Register responder's signature and certificate
        $\mathsf{state}[(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] \leftarrow \mathtt{registered}$.
        send InitSign to $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \mathtt{signer})$;
        wait for $(\mathtt{InitSign}, \mathsf{success}, pk)$; $\mathsf{pk} \leftarrow pk$;
        send responsively $(\mathtt{Register}, \mathsf{pk})$ to NET;
        wait for $(\mathtt{Register}, \mathsf{pk}, pid_{\mathtt{CA}})$ s.t. $pid_{\mathtt{CA}} \in \{0,1\}^*$;
        $\mathsf{pid}_{\mathtt{CA}} \leftarrow pid_{\mathtt{CA}}$;
        send $(\mathtt{Register}, \mathsf{pk}, pid_{\mathtt{CA}})$ to $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \varepsilon, \mathcal{F}_{\text{PKI}}^{\text{acc}} : \mathtt{client})$.[a]    {Register identity at $\mathcal{F}_{\text{PKI}}^{\text{acc}}$
    recv InitKE from I/O s.t. $\mathsf{state} = \mathtt{registered}$:    {Start KE.
        $\mathsf{state} \leftarrow \mathtt{started}$.
        send $(\mathtt{InitKE}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}))$ to NET.    {Notify network that the key exchange has started and the responder is ready to receive the first message.
    recv $(\mathtt{Receive}, \mathsf{initiator}, h_{init})$ from NET s.t. $\mathsf{state} = \mathtt{started}$:    {Receive first message, send second message.
        $\mathsf{h}_{\mathsf{init}} \leftarrow h_{init}$
        Choose $\mathsf{e}_{\mathsf{resp}} \leftarrow \mathbb{Z}_n$ uniformly at random, compute $h_{resp} = \mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}$.
        send $(\mathtt{Sign}, (\mathsf{h}_{\mathsf{init}}, \mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}, \mathsf{initiator}))$ to $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \mathtt{signer})$;
        wait for $(\mathtt{Signature}, \sigma)$.
        $\mathsf{state} \leftarrow \mathtt{inSession}$
        send $(\mathtt{Send}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}, \sigma)$ to NET.
    recv $(\mathtt{Receive}, \sigma)$ from NET s.t. $\mathsf{state} = \mathtt{inSession}$:    {Receive third message, output key.
        send responsively $(\mathtt{RetrieveCA}, \mathsf{initiator})$ to NET    {Query $\mathcal{A}$ for initiator's CA;
        wait for $(\mathtt{RetrieveCA}, \mathsf{initiator}, pid_{\mathtt{CA}})$;
        send $(\mathtt{Retrieve}, \mathsf{initiator}, pid_{\mathtt{CA}})$ to $((\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \varepsilon, \mathcal{F}_{\text{PKI}}^{\text{acc}} : \mathtt{client})$;
        wait for $(\mathtt{Retrieve}, \mathsf{initiator}, pid_{\mathtt{CA}}, pk)$.    {Get public verification key of intended partner.
        if $pk = \perp$:
            abort.
        send $(\mathtt{Verify}, (\mathsf{g}^{\mathsf{e}_{\mathsf{resp}}}, \mathsf{h}_{\mathsf{init}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})), \sigma, pk)$ to $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{initiator}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\text{sig}} : \mathtt{verifier})$;
        wait for $(\mathtt{VerResult}, b)$.
        if $b = \mathtt{false}$:
            abort.
        $\mathsf{sessionKey} \leftarrow \mathsf{h}_{\mathsf{init}}{}^{\mathsf{e}_{\mathsf{resp}}}$; $\mathsf{state} \leftarrow \mathtt{finished}$.
        send $(\mathtt{FinishKE}, \mathsf{sessionKey})$ to caller.
```

[a]We consider here one instance of the PKI, namely the one with SID ε.

Fig. 38: A real key exchange protocol $\mathcal{P}_{\text{iso}}^{\text{acc}}$ for realizing $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ (part 3).

Description of the initialization machine $\mathcal{F}_{\text{init}}^{\text{iso}} = (\texttt{init})$:

Participating roles: $\{\texttt{init}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{init}}$:

```
Implemented role(s): $\{\texttt{init}\}$
Subroutines: $\mathcal{P}_{\text{iso}}^{\text{acc}}$ : initiator, $\mathcal{P}_{\text{iso}}^{\text{acc}}$ : responder
Internal state:
    – init : $(\{0,1\}^*)^3 \to \{\texttt{true}, \texttt{false}\}$        {Initialization status of the machine, initially false
    – initiator : $(\{0,1\}^*)^3$        {The initiator of the key exchange
    – responder : $(\{0,1\}^*)^3$        {The responder of the key exchange
    – judge$_{\text{initiator}}$ : $(\{0,1\}^*)^3$        {The initiator's local judge
    – judge$_{\text{responder}}$ : $(\{0,1\}^*)^3$        {The responder's local judge
    – caller $\in (\{0,1\}^*)^3 \cup \{\perp\} = \perp$        {Stores the callers.
CheckID($pid, sid, role$):
    Accept all messages for the same $sid$.
Initialization:
    Parse sid$_{\text{cur}}$ as $(sid', pid_i, pid_r)$.
    initiator $\leftarrow (pid_i, (sid', pid_i, pid_r), \texttt{initiator})$.        {Store initiator and responder
    responder $\leftarrow (pid_r, (sid', pid_i, pid_r), \texttt{responder})$.
    judge$_{\text{initiator}}$ $\leftarrow ((\texttt{local}, pid_i, \texttt{initiator}), (sid', pid_i, pid_r), \texttt{judge})$.
    judge$_{\text{responder}}$ $\leftarrow ((\texttt{local}, pid_r, \texttt{responder}), (sid', pid_i, pid_r), \texttt{judge})$.        {Also store their judges entities
    caller $\leftarrow$ (pid$_{\text{call}}$, sid$_{\text{call}}$, role$_{\text{call}}$)
MessagePreprocessing:
    if (pid$_{\text{call}}$, sid$_{\text{call}}$, role$_{\text{call}}$) = initiator:
        init[initiator] $\leftarrow$ true
    else if (pid$_{\text{call}}$, sid$_{\text{call}}$, role$_{\text{call}}$) = responder:
        init[responder] $\leftarrow$ true
    else if (pid$_{\text{call}}$, sid$_{\text{call}}$, role$_{\text{call}}$) = judge$_{\text{initiator}}$:
        init[judge$_{\text{initiator}}$] $\leftarrow$ true
    else if (pid$_{\text{call}}$, sid$_{\text{call}}$, role$_{\text{call}}$) = judge$_{\text{responder}}$:
        init[judge$_{\text{responder}}$] $\leftarrow$ true
Main:
    recv initPartner from responder $\vee$ initiator:        {Request to initialize
        if init[initiator] = $\perp$:
            init(initiator)        {Trigger initialization of initiator
        else if init[responder] = $\perp$:
            init(responder)        {Trigger initialization of responder
        else if init[judge$_{\text{initiator}}$] = $\perp$:
            init(judge$_{\text{initiator}}$)        {Trigger initialization of judge$_{\text{initiator}}$
        else if init[judge$_{\text{responder}}$] = $\perp$:
            init(judge$_{\text{responder}}$)        {Trigger initialization of judge$_{\text{responder}}$
        else:        {All parties are initialized
            send initPartner to caller
    recv InitEntityDone from initiator $\vee$ responder $\vee$ judge$_{\text{initiator}}$ $\vee$ judge$_{\text{responder}}$:
        if init[initiator] = $\perp$:
            init(initiator)        {Trigger initialization of initiator
        else if init[responder] = $\perp$:
            init(responder)        {Trigger initialization of responder
        else if init[judge$_{\text{initiator}}$] = $\perp$:
            init(judge$_{\text{initiator}}$)        {Trigger initialization of judge$_{\text{initiator}}$
        else if init[judge$_{\text{responder}}$] = $\perp$:
            init(judge$_{\text{responder}}$)        {Trigger initialization of judge$_{\text{responder}}$
        else:        {All parties are initialized
            send initPartner to caller
```

Fig. 39: The initialization ITM $\mathcal{F}_{\text{init}}^{\text{iso}}$.

Description of $\mathcal{P}^{\text{iso}}_{\text{judge}} = (\texttt{judge})$:

```
Participating roles: {judge}
Corruption model:  incorruptible
```

Description of $M_{\texttt{judge}}$:

```
Implemented role(s): {judge}
Subroutines: $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ : judge, $\mathcal{P}^{\text{acc}}_{\text{iso}}$
Internal state:
    – initiator : $(\{0,1\}^*)^3$                                   {The initiator of the key exchange
    – responder : $(\{0,1\}^*)^3$                                   {The responder of the key exchange
CheckID($pid, sid, role$):
    Accept all entities of form $((\texttt{local}, pid_i, \texttt{initiator}), (sid', pid_i, pid_r), \texttt{judge})$ and $((\texttt{local}, pid_r, \texttt{responder}),$
    $(sid', pid_i, pid_r), \texttt{judge})$.
Initialization:
        Parse sid_cur as $(sid', pid_i, pid_r)$.
        initiator $\leftarrow (pid_i, (sid', pid_i, pid_r), \texttt{initiator})$.      {Store initiator and responder
        responder $\leftarrow (pid_r, (sid', pid_i, pid_r), \texttt{responder})$.
        if ITM was not activated via init macro, resp. a InitEntity message:
            send initPartner to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}^{\text{iso}}_{\text{init}} : \texttt{init})$
            wait for initPartner
Corruption behavior:
    – AllowCorruption($pid, sid, role$) :
        Parse $pid$ as $(pid', role')$
        $corr \leftarrow \textbf{corr}(pid', sid, role')$
        if $corr$:
            return true
        else:
            return false
    – DetermineCorrStatus($pid, sid, role$) :      {Consider local judge corrupted if associated client is corrupted.
        Parse $pid$ as $(pid', role')$
        $corr \leftarrow \textbf{corr}(pid', sid, role')$
        if $corr$:
            return true
        else:
            return false
Main:
        recv GetVerdict from I/O:                   {Forward local verdict from $\mathcal{F}^{\text{acc}}_{\text{PKI}}$
            send InitMe to initiator
            wait for _
            send InitMe to responder
            wait for _
            Parse pid_cur as $(pid', role')$.
            send GetVerdict to $((\text{initiator}, \texttt{client}), \varepsilon, \mathcal{F}^{\text{acc}}_{\text{PKI}} : \texttt{judge})$     {Query for the initiator's status at $\mathcal{F}^{\text{acc}}_{\text{PKI}}$
            wait for $(\texttt{GetVerdict}, v_1)$
            send GetVerdict to $((\text{responder}, \texttt{client}), \varepsilon, \mathcal{F}^{\text{acc}}_{\text{PKI}} : \texttt{judge})$     {Query for the responder's status at $\mathcal{F}^{\text{acc}}_{\text{PKI}}$
            wait for $(\texttt{GetVerdict}, v_2)$
            if $v_1 \neq \varepsilon \wedge role' = \text{responder}$:
                $v_1$.add(dis(initiator))          {The judge cannot be shure whether the intended partners verdict is trustful
            if $v_2 \neq \varepsilon \wedge role' = \text{initiator}$:
                $v_2$.add(dis(responder))          {The judge cannot be shure whether the intended partners verdict is trustful
            Let $verdict$ be
                1. $v_1$, if $role' = \texttt{responder}$
                2. $v_2$, if $role' = \texttt{initiator}$
            reply $(\texttt{GetVerdict}, verdict)$
        recv $(\texttt{GetJudicialReport}, msg)$ from I/O:     {Forward judicial report from lower level.
            reply $(\texttt{GetJudicialReport}, \varepsilon)$
```

Fig. 40: The judging functionality $\mathcal{P}^{\text{iso}}_{\text{judge}}$ for $\mathcal{P}^{\text{acc}}_{\text{iso}}$.

Description of $\mathcal{P}^{\text{iso}}_{\text{sv}} = (\texttt{supervisor})$:

```
Participating roles: {supervisor}
Corruption model:    incorruptible
```

Description of $M_{\texttt{supervisor}}$:

```
Implemented role(s): {supervisor}
Subroutines: $\mathcal{F}^{\text{acc}}_{\text{PKI}}$ : supervisor
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
        recv (IsAssumptionBroken?, $msg, id$):
            reply (IsAssumptionBroken?, false)
        recv (corruptInt?, $pid, sid, role$) s.t. $role \notin$ {initiator, responder, judge, supervisor}:
            if $role$ = client:
                $corrRes \leftarrow$ corr($pid, sid, \mathcal{F}^{\text{acc}}_{\text{PKI}}$ : client)        {Request corruption status at $\mathcal{F}^{\text{acc}}_{\text{PKI}}$
                reply (corruptInt, $corrRes$)
            else:
                send (corruptInt?, $pid, sid, role$) to ($\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\text{acc}}_{\text{PKI}}$ : supervisor)
                wait for (corruptInt?, $corrRes$)
                reply (corruptInt, $corrRes$)
```

Fig. 41: The supervisor $\mathcal{P}^{\text{iso}}_{\text{sv}}$ for $\mathcal{P}^{\text{acc}}_{\text{iso}}$.

the name pid of a malicious party that caused the protocol to abort (cf., [14, 34, 55]). Depending on this choice, $\mathcal{F}^{\text{id-ab}}_{\text{MPC}}$ either returns the outputs of f or a special message $(\texttt{abort}, pid)$ to the honest parties.

Capturing identifiable abort using AUC. We can easily capture the same properties as $\mathcal{F}^{\text{id-ab}}_{\text{MPC}}$ by applying AUC to the basic functionality $\mathcal{F}_{\text{MPC}}$. We set $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{correctness}\}$ and, as part of the transformation $\mathcal{T}_2$, redefine the behavior of $\mathcal{F}_{\text{MPC}}$ to output abort, instead of the actual function output, iff correctness is broken for that party. To also capture the same relationships and exact accountability level required by identifiable abort, we instantiate $\mathcal{F}_{\text{judgeParams}}$ to impose the following additional requirements whenever the simulator tries to break correctness: *(i)* no honest party has already obtained an output, *(ii)* if correctness is broken for one honest party, it must be broken for all others at the same time, and *(iii)* all verdicts for local judges of honest parties are identical and of the form $\mathsf{dis}(A)$ for some party A.

The resulting functionality $\mathcal{F}^{\text{acc}}_{\text{MPC}}$ models the exact same properties as $\mathcal{F}^{\text{id-ab}}_{\text{MPC}}$, with the only syntactical difference being that $\mathcal{F}^{\text{id-ab}}_{\text{MPC}}$ includes the verdict as part of the protocol output while in $\mathcal{F}^{\text{acc}}_{\text{MPC}}$ the verdict is obtained separately from a judge.

H. Further Details on MPC Accountability Properties via AUC

In this section, we detail our presentation from Appendix G and show that AUC can be seen as a generalization of the existing accountability related MPC/UC literature. Therefore, we firstly illustrate in detail how the common MPC property of (internal) identifiable abort translates to AUC. We use a standard MPC functionality $\mathcal{F}_{\text{MPC}}$ without in-build identifiable abort and depict how identifiable abort is typically captured and how one can capture it with AUC. We then briefly discuss that this approach also works for other common properties, such as *verifiability*.

The ideal MPC functionality $\mathcal{F}_{\text{MPC}}$. For common ideal MPC functionalities (cf., e.g., [8, 21, 24]) one assumes that there are m parties involved in the computation, each party p_i contributes some secret input x_i, and the parties want to distributedly compute the output of a function f. The ideal MPC functionality $\mathcal{F}_{\text{MPC}}$ models a trusted third party which executes the computation on behalf of the participating users. Within $\mathcal{F}_{\text{MPC}}$, the adversary $\mathcal{A}$ is typically allowed to (statically) corrupt parties. The execution of the computation via $\mathcal{F}_{\text{MPC}}$ is separated in phases. All parties are expected/forced to be in the same phase of the protocol: *(i) Input:* Each honest party inputs x_i to $\mathcal{F}_{\text{MPC}}$. Parties controlled by the adversary could input tainted data x_i'. *(ii) Computation:* $\mathcal{F}_{\text{MPC}}$ evaluates f given the inputs from the parties. *(iii) Finalization:* $\mathcal{F}_{\text{MPC}}$ provides the output of the computation to the parties. $\mathcal{F}_{\text{MPC}}$ typically guarantees the correct computation of the function f. We present a formal specification of $\mathcal{F}_{\text{MPC}}$ in Figure 42 where the highlighted code denotes the differences between $\mathcal{F}_{\text{MPC}}$ and $\mathcal{F}^{\text{acc}}_{\text{MPC}}$.

Identifiable abort in UC literature. With identifiable abort (cf., e.g., [14, 34, 55]), one typically protects *correctness* of a MPC protocol and in particular the correctness of the protocol's output. This models that parties abort the protocol and blame a misbehaving party if, e. g., *(i)* a party identifies a misbehaving party which tries to distort the computation or *(ii)* a (corrupted) party aborts the protocol. We focus on internal/local identifiable abort, i. e., we interpret identifiable abort as a *local* accountability property. This is often the case in MPC protocols as verdicts often rely on information that is only accessible to protocol participants and thus do not convince outsiders of the protocol.

An adapted ideal MPC functionality $\mathcal{F}'_{\mathrm{MPC}}$ which covers identifiable abort typically differs from $\mathcal{F}_{\mathrm{MPC}}$ in the following details: *(i)* Honest parties do not receive the output of the computation in case of an abort. *(ii)* To abort, a corrupted party p inputs the string `abort` to $\mathcal{F}'_{\mathrm{MPC}}$, this informs $\mathcal{F}'_{\mathrm{MPC}}$ that $\mathcal{A}$, resp. party p, aborted the execution of the protocol. *(iii)* $\mathcal{F}_{\mathrm{MPC}}$ then typically informs honest parties that p aborted the protocol. This "individual verdict" on p allows the honest parties in the MPC protocol to exclude misbehaving parties from the next computation attempt.

Identifiable Abort via AUC. In contrast to Appendix G, we do not consider the traditional interpretation of MPC protocols which assumes that all parties do all phases of the MPC protocol synchronously – we call this case synchronous MPC. We focus here on what we call the "asynchronous" case, where parties are not forced to be in the same phase of the protocol. However, we provide a full specification of the synchronous variant of $\mathcal{F}_{\mathrm{MPC}}$ which we call $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$ in Figure 42 and 43 which include the properties of identifiable abort, called `non-aborting`, verifiability, called `correctness`, and `privacy`. In what follows, we focus on the asynchronous variant of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$ and include identifiable abort to the functionality. We call this functionality $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ (cf. Figure 44 and 45). In $\mathcal{F}_{\mathrm{MPC}}$, the following major adaptions are necessary to derive $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ (besides mapping messages): *(i)* Apply the AUC transformation $\mathcal{T}_1(\cdot)$ to $\mathcal{F}_{\mathrm{MPC}}$, *(ii)* add one local judge per participant/node to $\mathcal{F}_{\mathrm{MPC}}$, i. e., $\mathsf{pids}_{\mathsf{judge}} = \{\mathtt{local}\} \times \{0,1\}^*$ and every set member is meant to be of form $(\mathtt{local}, pid, role)$ ($\mathsf{ids}_{\mathsf{assumption}} = \emptyset$ as we do not consider assumption-based properties here), *(iii)* add the property `correctness` to $\mathsf{Sec}^{\mathrm{acc}}$ in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ and $\mathcal{F}^{\mathrm{async\text{-}MPC}}_{\mathrm{judgeParams}}$.[18] *(iv)* Allow $\mathcal{A}$ to abort the computation using the `BreakAccProp` command. $\mathcal{A}$'s request needs to mark correctness broken for honest parties and needs to include a (fair) individual verdict for each of the parties. *(v)* $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ outputs `abort` to a party if `correctness` is broken for this party. Parties can query their judge to retrieve the verdict which includes who aborted the protocol. *(vi)* We adapt $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$, resp. $\mathcal{F}^{\mathrm{async\text{-}MPC}}_{\mathrm{judgeParams}}$, such that they ensure that *(a)* aborting is not possible after the first honest party received output from $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ and vice versa, *(b)* that no party receives an output if one party received `abort`. *(vii)* $\mathcal{F}^{\mathrm{async\text{-}MPC}}_{\mathrm{judgeParams}}$ enforces that (local) verdicts are available via the internal local judges, and that *(viii)* all internal judges output the same verdict.

Clearly, the AUC formulation of identifiable abort can be mapped to the traditional definition of identifiable abort: As soon as $\mathcal{A}$ signalizes that a corrupted party will abort the protocol, all honest parties will be informed that the protocol aborted and which party caused the abort. This statement holds true for $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$ as well as for $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$. Of course one can enhance our approach above analogously to achieve *public* identifiable abort (cf., e.g., [30, 60].

Verifiability. Verifiability in MPC is also a correctness property [35, 84, 86]: in the case of public or universal verifiability, every party including protocol outsiders should be able to verify that the MPC's output was computed correctly or honestly. Due to the privacy guarantees of MPC protocols, public verifiability typically only provides a weak level of accountability as one cannot identify misbehaving parties individually.

The common formalization of verifiability is analogous to the formalization of identifiable abort. For public verifiability (cf. [6]), one sometimes expects some *certificate of misbehavior* created by a judge which allows outsiders to verify that the blamed party indeed misbehaved.

To capture verifiability with AUC, one uses the same techniques as for identifiable abort. The major difference between both approaches are *(i)* verifiability is a public property, thus restrictions are incorporated into a public judge instead of local judges, *(ii)* $\mathcal{F}^{\mathrm{async\text{-}MPC}}_{\mathrm{judgeParams}}$ does not impose restrictions on the verdict, and *(iii)* one may use the judicial report defined at $\mathcal{F}^{\mathrm{async\text{-}MPC}}_{\mathrm{judgeParams}}$ to output a certificate of misbehavior.

For other MPC accountability properties, such as *auditability* [13], *openability* [34] or *privacy* [8], $\mathcal{A}$ also signalizes the ideal functionality that a property is broken and the ideal functionality then acts appropriately which typically signalizes that a party or a group of parties misbehaved. The approach above works analogously for other accountability properties from MPC. Thus, AUC can be seen as generalization of existing work regarding accountability in MPC/UC literature.

[18]We note that `correctness` captures identifiable abort in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{async\text{-}MPC}}$ while it captures verifiability in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$. For the sake of presentation fot $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$ we want to keep naming in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$ as close to literature as possible.

Description of the protocol $\mathcal{F}^{\text{acc}}_{\text{MPC}} = (\texttt{node}, \texttt{judge}, \texttt{supervisor})$

```
Participating roles: {node, judge, supervisor}
Corruption model: static
Protocol parameters:
    – $n \in \mathbb{N}$                                              {Input length for the MPC function
    – $f : (\{0,1\}^*)^{|\mathsf{P}|} \to (\{0,1\}^*)^{|\mathsf{P}|}$, s.t. $f = (f_1, \ldots, f_{|\mathsf{P}|})$      {the MPC function
    – $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$        {Accountability properties, expected here: {privacy, correctness, non-aborting}
    – $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$        {Assumption-based security properties, expected here: $\emptyset$
    – $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$    {set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)
    – $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$          {set of entities/IDs where properties are ensured via assumptions
```

Description of $M_{\texttt{node}}$:

```
Implemented role(s): {node}
Subroutines: $\mathcal{F}^{\text{MPC}}_{\text{judgeParams}}$ : judgeParams
Internal state:
    – $\mathsf{P} \subset \{0,1\}^*, \mathsf{P} \neq \emptyset, |\mathsf{P}| < \infty$                {Parties involved in the protocol
    – $\mathsf{inputs} : \mathsf{P} \to \{0,1\}^n$            {The inputs for the computation, initially always set to $\bot$
    – $\mathsf{outputs} : \mathsf{P} \to \{0,1\}^*$           {The outputs of the computation, initially always set to $\bot$
    – $\mathsf{state} \in \{\texttt{uninitialized}, \texttt{computation}, \texttt{finalized}\}$, $\mathsf{state} = \texttt{uninitialized}$     {State of the MPC computation
    – $\mathsf{corruptedIntParties} \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}$[a] $\cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$   {The set of corrupted internal parties $(pid, sid, role)$
    – $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\texttt{true}, \texttt{false}\}$  {Stores broken security assumptions per id, initially false $\forall$entries
    – $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\texttt{true}, \texttt{false}\}$   {Stores broken security properties per judge/id, initially false $\forall$entries
    – $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$        {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
CheckID($pid, sid, role$):
    Accept all messages if $pid \in \mathsf{P}$ and with same $sid$.     {A single instance manages all parties in a single session.
Corruption behavior:
    – AllowCorruption($pid, sid, role$):
        Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
        if $role = \texttt{judge}$:
            send $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
             to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$      {$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted
            wait for $b$; return $b$
    – DetermineCorrStatus($pid, sid, role$):
        if $role = \texttt{judge}$:                  {$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status
            send $(\texttt{CorruptionStatus?}(pid, sid, \texttt{judge}), \mathsf{internalState})$
             to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
            wait for $b$; return $b$
    – AllowAdvMessage($pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m$)      {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}^{\text{acc-cp}}_{\text{judgeParams}}$
        Do not allow sending messages to $\mathcal{F}_{\text{judgeParams}}$.
Initialization:
        send responsively InitMe to NET                 {$\mathcal{A}$ determines the involved parties
        wait for $(\texttt{Init}, P)$ s.t. $P \subset \{0,1\}^*$
        $\mathsf{P} \leftarrow P$
MessagePreprocessing:
        if $\mathsf{state} = \texttt{uninitialized} \wedge$ number of inputs that are $\neq \bot$ is $|\mathsf{P}| \wedge$ brokenProps is false for all entries.:
                                                        {Update state/computation phase
            $\mathsf{state} \leftarrow \texttt{computation}$
Main:
        recv $(\texttt{Compute}, x)$ from I/O s.t. $\mathsf{pid}_{\mathsf{cur}} \in \mathsf{P} \wedge \mathsf{state} \in \{\texttt{uninitialized}\}$:     {Input from uncorrupted users
            $\mathsf{inputs}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow x$                        {Record user input
        recv $(\texttt{Compute}, pid, x)$ from NET s.t. $pid$ in CorruptionSet $\wedge\ pid \in \mathsf{P}, \mathsf{state} \in \{\texttt{uninitialized}, \texttt{computation}\}$:   {$\mathcal{A}$ input for corrupted parties
            $\mathsf{inputs}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow x$                        {Record user input
        recv Finalize from NET:
            if $\mathsf{state} = \texttt{computation} \wedge \forall pid \in \mathsf{P} : \mathsf{inputs}[pid] \neq \bot \wedge \mathsf{state} = \texttt{uninitialized}$:   {That is, there was no cheating/corrupting
                Let $w_i$ s.t. $(pid_i, w_i) \in \mathsf{inputs}, i \in \{1, \ldots, |\mathsf{P}|\}$
                $(y_1, \ldots, y_{|\mathsf{P}|}) \leftarrow f(w_1, \ldots, w_{|\mathsf{P}|})$       {If there was no "interrupt" from $\mathcal{A}$, $f$ is computed as expected
                for $i = 1$ to $|\mathsf{P}|$ do:
                    $\mathsf{outputs}[pid_i] \leftarrow y_i$
                $\mathsf{state} \leftarrow \texttt{finalized}$
                Let $output$ contain every $\mathsf{outputs}[pid]$ where $pid$ is in CorruptionSet.
                reply $(\texttt{Finalize}, output)$                {Send output of corrupted parties to $\mathcal{A}$
        recv GetResult from I/O s.t. $\mathsf{pid}_{\mathsf{cur}} \in \mathsf{P}$:
            if $\mathsf{brokenProps}[\texttt{non-aborting}, (\texttt{local}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})]$:       {We are in the identifiable abort case
                reply $(\texttt{GetResult}, \texttt{abort}, \mathsf{verdicts})$          {Output abort information and verdict
            else if $\mathsf{brokenProps}[\texttt{privacy}, (\texttt{local}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})]$:     {We are in the case that $\mathcal{A}$ broke privacy
             reply $(\texttt{GetResult}, \texttt{corrupted}, \mathsf{verdicts})$          {Output privacy break and verdict
            else if $\mathsf{brokenProps}[\texttt{correctness}, (\texttt{local}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})]$:   {We are in the case that $\mathcal{A}$ broke correctness of the calculation
             reply $(\texttt{GetResult}, \texttt{correctness}, \mathsf{verdicts})$        {Output correctness break and verdict
            reply $(\texttt{GetResult}, \mathsf{outputs}[\mathsf{pid}_{\mathsf{cur}}])$
          Include the missing static code from the AUC transformation $\mathcal{T}_1(\cdot)$ here, i.e., include additional code from Figure 2 and 3 here.
```

[a]Here: $\mathsf{Roles}_{\mathcal{F}} := \{\texttt{node}\}$.

Fig. 42: The MPC functionality $\mathcal{F}^{\text{acc}}_{\text{MPC}}$.

Description of $\mathcal{F}^{\mathrm{MPC}}_{\mathrm{judgeParams}} = (\texttt{judgeParams})$:

```
Participating roles: {judgeParams}
Corruption model:   incorruptible
```

Description of $M_{\texttt{judgeParams}}$:

```
Implemented role(s): {judgeParams}
CheckID($pid, sid, role$):
    Accept all messages with the same $sid$.
Main:
    recv (BreakAccProp, $verdict, toBreak, internalState$) from I/O:
        if $toBreak = \{\texttt{non-aborting}\} \times (\{\texttt{local}\} \times \{0,1\}^{2*}) \wedge state \in \{\texttt{uninitialized}, \texttt{compute}\}$:    {$\mathcal{A}$ aborts computation
            reply (BreakAccProp, true, $\varepsilon$)
        if $toBreak = \{\texttt{correctness}\} \times (\{\texttt{local}\} \times \{0,1\}^{2*}) \wedge state = \texttt{compute}$:    {This models a cheat attempt
            reply (BreakAccProp, true, $\varepsilon$)
        if $toBreak = \{\texttt{privacy}\} \times (\{\texttt{local}\} \times \{0,1\}^{2*}) \wedge state = \texttt{compute}$:    {This models an attempt to break input privacy
            Let $\bar{\omega}$ be the vector of all values from inputs.
            if eval($verdict$) = true:    {Need to check whether $verdict$ evaluates to true as leakage forwarded to $\mathcal{A}$ anyways
                reply (BreakAccProp, true, $\bar{\omega}$)
            else:
                reply (BreakAccProp, false, $\varepsilon$)
        else:
            reply (BreakAccProp, false, $\varepsilon$)
    recv (GetJudicialReport, $msg, internalState$) from I/O:    {Generate judicial report
        reply (GetJudicialReport, $\varepsilon$)    {Return state variable as report
    recv (Corrupt, ($id, sid$, judge), $internalState$) from I/O:    {$\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$ allows to corrupt a local judge iff the accompanied initiator or responder is corrupted
        if $id$ = (local, $pid, role$) $\wedge$ ($pid, sid, role$) $\in$ CorruptionSet:
            reply true
        else:
            reply false
    recv (CorruptionStatus?, ($id, sid$, judge), $internalState$) from I/O:    {$\mathcal{F}^{\mathrm{KE}}_{\mathrm{judgeParams}}$ interprets a local judge as corrupted iff the accompanied initiator or responder is corrupted
        if ($id, sid$, judge) $\in$ CorruptionSet:
            reply true
        else if $id$ = (local, $pid, role$) $\wedge$ ($pid, sid, role$) $\in$ CorruptionSet :
            reply true
        else:
            reply false
```

Fig. 43: The judge parameter functionality $\mathcal{F}^{\mathrm{MPC}}_{\mathrm{judgeParams}}$ for $\mathcal{F}^{\mathrm{acc}}_{\mathrm{MPC}}$.

Description of the protocol $\mathcal{F}^{\text{acc}}_{\text{async- MPC}} = (\texttt{node}, \texttt{judge}, \texttt{supervisor})$

```
Participating roles: $\{\texttt{node}, \texttt{judge}, \texttt{supervisor}\}$
Corruption model: static
Protocol parameters:
    – $n \in \mathbb{N}, f : (\{0,1\}^*)^{|\mathsf{P}|} \to (\{0,1\}^*)^{|\mathsf{P}|}$, s.t. $f = (f_1, \ldots, f_{|\mathsf{P}|})$    {Input length for the MPC function and the function itself
    – $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$    {Accountability properties
    – $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$    {Assumption-based security properties
    – $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$    {set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)
    – $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$    {set of entities/IDs where properties are ensured via assumptions
```

Description of $M_{\texttt{node}}$:

```
Implemented role(s): $\{\texttt{node}\}$
Subroutines: $\mathcal{F}^{\text{MPC}}_{\text{judgeParams}}$ : judgeParams
Internal state:
    – $\mathsf{P} \subset \{0,1\}^*, \mathsf{P} \neq \emptyset, |\mathsf{P}| < \infty$    {Parties involved in the protocol
    – $\mathsf{inputs} : \mathsf{P} \to \{0,1\}^n$    {The inputs for the computation, initially always set to $\bot$
    – $\mathsf{outputs} : \mathsf{P} \to \{0,1\}^*$    {The outputs of the computation, initially always set to $\bot$
    – $\mathsf{state} : \mathsf{P} \to \{\texttt{uninitialized}, \texttt{computation}, \texttt{finalized}, \texttt{finishedOutput}\}$
                                            {State of the MPC computation per party, initillay uninitialized
    – $\mathsf{corruptedIntParties} \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$    {The set of corrupted internal parties $(pid, sid, role)$
    – $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\texttt{true}, \texttt{false}\}$ {Stores broken security assumptions per id, initially false $\forall$entries
    – $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\texttt{true}, \texttt{false}\}$    {Stores broken security properties per judge/id, initially false $\forall$entries
    – $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$    {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
CheckID($pid, sid, role$):
        Accept all messages if $pid \in \mathsf{P}$ and with same $sid$.    {A single instance manages all parties in a single session.
Additional Corruption behavior:
    – AllowCorruption($pid, sid, role$):
        Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
        if $role = \texttt{judge}$:
            send $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$ to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$    {$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted
            wait for $b$; return $b$
    – DetermineCorrStatus($pid, sid, role$):
        if $role = \texttt{judge}$:    {$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status
            send $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$ to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
            wait for $b$; return $b$
    – AllowAdvMessage($pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m$)
        Do not allow sending messages to $\mathcal{F}_{\text{judgeParams}}$.    {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}_{\text{judgeParams}}$ in the name of corrupted parties.
Initialization:
        send responsively InitMe to NET    {$\mathcal{A}$ determines the involved parties
        wait for $(\texttt{Init}, P)$ s.t. $P \subset \{0,1\}^*$
        $\mathsf{P} \leftarrow P$
Main:
        recv $(\texttt{Compute}, x)$ from I/O s.t. $\mathsf{pid}_{\mathsf{cur}} \in \mathsf{P} \land \mathsf{state} \in \{\texttt{uninitialized}\}$:    {Input from uncorrupted users
            $\mathsf{state}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow \texttt{computation}$; $\mathsf{inputs}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow x$    {Update phase of $\mathsf{pid}_{\mathsf{cur}}$ and record user input

        recv $(\texttt{Compute}, pid, x)$ from NET s.t. $pid$ in CorruptionSet $\land\, pid \in \mathsf{P}, \mathsf{state} \in \{\texttt{uninitialized}, \texttt{computation}\}$:    {$\mathcal{A}$ input for corrupted parties
            $\mathsf{state}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow \texttt{computation}$    {Update phase of $\mathsf{pid}_{\mathsf{cur}}$
            $\mathsf{inputs}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow x$    {Record user input

        recv Finalize from NET:
            if $\forall pid \in \mathsf{P} : \mathsf{state}[pid] = \texttt{computation} \land \forall pid \in \mathsf{P} : \mathsf{inputs}[pid] \neq \bot$:    {Trigger computation of $f$
                Let $w_i$ s.t. $(pid_i, w_i) \in \mathsf{inputs}, i \in \{1, \ldots, |\mathsf{P}|\}$
                $(y_1, \ldots, y_{|\mathsf{P}|}) \leftarrow f(w_1, \ldots, w_{|\mathsf{P}|})$    {If there was no "interrupt" from $\mathcal{A}$, $f$ is computed as expected
                for $i = 1$ to $|\mathsf{P}|$ do:
                    $\mathsf{outputs}[pid_i] \leftarrow y_i$
                if There is a brokenProps of type correctness true:
                    $output \leftarrow \varepsilon$    {The protocol aborted, no one receives output
                else:
                    Let $output$ contain every $\mathsf{outputs}[pid]$ where $pid$ is in CorruptionSet.
                reply $(\texttt{Finalize}, output)$    {Send output of corrupted parties to $\mathcal{A}$

        recv $(\texttt{Finalize}, P)$ from NET s.t. $P \subset \mathsf{P}$:    {$\mathcal{A}$ may advance participants from phase computation to finalized
            for all $p \in P$ do:
                if $\mathsf{state}[p] = \texttt{computation}$: $\mathsf{state}[p] \leftarrow \texttt{finalized}$

        recv GetResult from I/O s.t. $\mathsf{pid}_{\mathsf{cur}} \in \mathsf{P}$:    {Access only for "registered" parties
            if $\mathsf{brokenProps}[(\texttt{correctness}, (\texttt{local}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})]$:    {Handle identifiable abort
                reply $(\texttt{GetResult}, \texttt{abort})$    {Output abort information
            else if $\mathsf{state}[\mathsf{pid}_{\mathsf{cur}}] \in \{\texttt{uninitialized}, \texttt{computation}\}$:    {Party did not finish computation so far
                reply $(\texttt{GetResult}, \bot)$
            else if There exists an entry $\mathsf{brokenProps}[(\texttt{correctness}, (\texttt{local}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})] = \texttt{true}$:    {If abort is recorded at another party
                reply $(\texttt{GetResult}, \bot)$    {Supress output in case of an abort of another party
            else if $\mathsf{state}[\mathsf{pid}_{\mathsf{cur}}] = \texttt{finalized}$:
                $\mathsf{state}[\mathsf{pid}_{\mathsf{cur}}] \leftarrow \texttt{finishedOutput}$
                reply $(\texttt{GetResult}, \mathsf{outputs}[\mathsf{pid}_{\mathsf{cur}}])$    {If $\mathcal{A}$ did not abort, output result

    Include the missing static code from the AUC transformation $\mathcal{T}_1(\cdot)$ here, i.e., include additional code from Figure 2 and 3 here.
```

[a]Here: $\mathsf{Roles}_{\mathcal{F}} := \{\texttt{node}\}$.

Fig. 44: The asynchronous MPC functionality $\mathcal{F}^{\text{acc}}_{\text{async- MPC}}$.

Description of $\mathcal{F}^{\text{async- MPC}}_{\text{judgeParams}} = (\texttt{judgeParams})$:

Participating roles: $\{\texttt{judgeParams}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{judgeParams}}$:

```
Implemented role(s): {judgeParams}
CheckID(pid, sid, role):
    Accept all messages with the same sid.
Main:
    recv (BreakAccProp, $verdict$, $toBreak$, $internalState$) from I/O:
        Let $ids \subset \{0,1\}^*$ be the IDs part from $toBreak$.
        if $toBreak \subset \{\texttt{correctness} \times ids\} \wedge \nexists p \in \mathsf{P}$ (P extracted from $internalState$) with $\mathsf{state}[p] = \texttt{finishedOutput} \wedge$
          $[\forall id_1, id_2$, such that $id_1 \neq id_2$ where $verdict[id_1] \neq \varepsilon$, resp. $verdict[id_2] \neq \varepsilon \wedge$
          it holds true that $verdict[id_1] = verdict[id_2]$ $\wedge$
          all verdicts in $verdict$ are individual verdicts:        {$\mathcal{A}$ aborts computation, only allowed if restrictions are met
            reply (BreakAccProp, true, $\varepsilon$)                {Accept abort
        else:
            reply (BreakAccProp, false, $\varepsilon$)               {Decline abort
    recv (GetJudicialReport, $msg$, $internalState$) from I/O:       {Generate judicial report
        reply (GetJudicialReport, $\varepsilon$)                     {Return state variable as report
    recv (Corrupt, ($id$, $sid$, judge), $internalState$) from I/O:  {$\mathcal{F}^{\text{KE}}_{\text{judgeParams}}$ allows to corrupt a local judge iff the
        if $id = (\texttt{local}, pid, role) \wedge (pid, sid, role) \in$ CorruptionSet:   accompanied initiator or responder is corrupted
            reply true
        else:
            reply false
    recv (CorruptionStatus?, ($id$, $sid$, judge), $internalState$) from I/O:   {$\mathcal{F}^{\text{KE}}_{\text{judgeParams}}$ interprets a local judge as corrupted iff
        if ($id$, $sid$, judge) $\in$ CorruptionSet:                 the accompanied initiator or responder is corrupted
            reply true
        else if $id = (\texttt{local}, pid, role) \wedge (pid, sid, role) \in$ CorruptionSet :
            reply true
        else:
            reply false
```

Fig. 45: The judge parameter functionality $\mathcal{F}^{\text{async- MPC}}_{\text{judgeParams}}$ for $\mathcal{F}^{\text{acc}}_{\text{async- MPC}}$.