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.

I. Introduction

Accountability is a prominent concept that is widely used in security. Many security properties and applications, such as auctions [6], [28], [94], e-voting [65], [63], [1], non-repudiation [95], [96], [97], multi-party computation (MPC) [53], [7], [5], [12], public key infrastructures (PKIs) [70], [73], [59], distributed ledgers [48], [58], [15], [79], [38], [14], [36], [45], [85], DRM [90], [88], power infrastructures [57], [72], content delivery networks [29], and distributed systems [87], [51], [91] 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., [65], [61], [76], [60], [66], [31], [41], [40], [42], [43], [51], [56]), which we therefore also consider in this work and often just call *accountability* (see also Section II-G). 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., [65], [44], [60]). 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* [65], 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) [47], [39]; 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. [43]), which includes many special cases such as *proactive security* [11], [2]. 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. [48] 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-

DOI 10.1109/SP46215.2023.00107

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 [51] to strengthen the security property of *consistency* in Byzantine-fault tolerant (BFT) consensus protocols (e. g., [27], [80], [92]): 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., [21], [22], [62], [17], [52], [16]) 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., [12], [32], [71], [54], [23], [33], [13]). 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., [54], [13]), *(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 [65], [60], [76], [61]: *(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 [21], [22], including its variants (e.g., SUC [23], GUC [17]), the GNUC [52], the CC [74], and the IITM [62] 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 in the available space. 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], [48], the consensus service Hashgraph [10], and content delivery networks [37]. 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 [25], [64], [55], [16], 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 B 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 II presents AUC, including a discussion of its core concepts. Section III provides our case studies. We discuss related work in Section IV. Further details are given in the appendix. Full definitions and proofs are provided in our technical report [49].

II. AUC – Accountable Universal Composability

In this section, we introduce the accountable universal composability (AUC) framework. We first clarify notation and terminology in Section II-A. Section II-B discusses several high-level ideas of AUC before we formally specify the AUC transformation for ideal functionalities in Section II-C, with AUC's modeling of real protocols covered in Section II-D. In Section II-E, we discuss several aspects of AUC's composability abilities. In Section II-F, we present a deterrence analysis to analyze the behavior of rational adversaries. Section II-G concludes this section with a discussion on AUC.

A. Notation and Terminology

Computational Model. Formally, we define AUC within the iUC model [16], a recent model for universal composability. However, AUC and its concepts can also be used within existing other models for universal composability, e. g., [68], [16], [17], [8]. 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. In Appendix A, we provide an overview of the used pseudo code notation, which, however, should be mostly self explanatory.

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.

[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.

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.

B. 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:

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* [31]), 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 [34] 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 III-B 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.

C. 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 II-B, 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 accountability levels; we explain this later). For example, for local judges $\mathcal{F}_{\text{judgeParams}}$ 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.

[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)      {$\mathcal{F}_{\mathrm{judgeParams}}$
           to ($pid, sid, \mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)    decides whether
         wait for $b$; return $b$                                  judges can be
                                                                  corrupted
 – 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:
 – brokenProps : $(\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to$
   {true, false}
        {Stores broken security properties per judge/id, initially false ∀entries
 – verdicts : $\mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$       {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially $\varepsilon$
 – brokenAssumptions : $\mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to$ {true, false}
        {Stores broken security assumptions per id, initially false ∀entries
 – corruptedIntParties $\subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}$[b] $\cup$
   {judge, 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 $\mathcal{F}$.

```
Additional code for the judge roles:
recv (BreakAccProp, $verdict$, $toBreak$) from NET[a] to ($pid, sid$, judge)
  s.t. $toBreak \subseteq \mathsf{Sec}^{\mathsf{acc}} \times \mathsf{pids}_{\mathsf{judge}} \wedge verdict$ maps from $\mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$:
  ($successful, leakage$) ← breakAttempt($verdict, toBreak$)
                                       {breakAttempt is defined below
  reply (BreakAccProp, $successful, leakage$)
recv GetVerdict from I/O to ($pid_j, sid$, judge):   {The environment can
  reply (GetVerdict, verdicts[$pid_j$])               query the verdicts of
                                                      local and public judges
recv (GetJudicialReport, $msg$) from I/O to ($pid_j, sid$, judge):
        {The environment may query for local or public judicial reports
  send (GetJudicialReport, $msg$, internalState)     {Forward judicial report
    to ($pid_j, sid, \mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)    request to $\mathcal{F}_{\mathrm{judgeParams}}$
  wait for (GetJudicialReport, $report$)
  reply (GetJudicialReport, $report$)

Helperfunction:
procedure 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 dis(($pid, sid, role$)),
    2. it holds true that eval($verdict[pid_j]$) = true,[b]
  Check that $\forall (prop, pid_j) \in toBreak$:
    3. $verdict[pid_j] \neq \varepsilon$,
    4. brokenAssumptions[$prop, pid_j$] = 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(false, $\varepsilon$)
  send (BreakAccProp, $verdict, toBreak$, internalState)
    to (_, _, $\mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)
    {$\mathcal{F}_{\mathrm{judgeParams}}$ can impose further conditions, e.g., on verdicts or whether
     it is allowed to violate breakable security properties.
  wait for (BreakAccProp, $successful, leakage$)
  if $successful$:
    for all $\forall pid_j$ s.t. $verdict[pid_j] \neq \varepsilon$ do:     {Record accepted local
      verdicts[$pid_j$] ← $verdict[pid_j]$                         and public verdict
    for all ($prop, pid_j$) ∈ $toBreak$ do:
      brokenProps[$prop, pid_j$] ← 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 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 eval(ε) := true.

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

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 II-B 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.

Judges. Figure 2 specifies the code AUC adds to ideal functionalities to model judges. Each judge in AUC is an entity of the form $(pid_j, sid, \mathtt{judge})$ 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

```
Additional code for the supervisor role:
recv (BreakAssumption, $toBreak$) from NET to $(\_, \_, \texttt{supervisor})$
    s.t. $toBreak \subseteq \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}}$: {$\mathcal{A}$ may break these assumptions
    for all $(prop, id) \in toBreak$ do:
        brokenAssumptions$[prop, id] \leftarrow$ true          {Record broken assumptions
        if $prop \notin \mathsf{Sec}^{\mathsf{acc}} \vee id \notin \mathsf{pids}_{\mathsf{judge}}$:
            brokenProps$[prop, id] \leftarrow$ true
        {Record property as broken if not additionaly secured via accountability
    send (BreakAssumption, $toBreak$, internalState)
      to $(pid, sid, \mathcal{F}_{\mathrm{judgeParams}}$ : judgeParams)     {$\mathcal{F}_{\mathrm{judgeParams}}$ provides leakage
    wait for (BreakAssumption, $leakage$)
    reply (BreakAssumption, $leakage$)

recv (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
    corruptedIntParties.add($(pid, sid, role)$)
    reply (corruptInt, ack)

recv (IsAssumptionBroken?, $prop, id$) from I/O                    {The environment may ask
    to $(\_, \_, \texttt{supervisor})$ s.t. $id \in \mathsf{ids}_{\mathsf{assumption}}$:            whether a local or public
    if $prop \in \mathsf{Sec}^{\mathsf{assumption}}$:                                          properties are broken
        reply (IsAssumptionBroken?, brokenAssumptions$[prop, id]$)
    else:
        reply (IsAssumptionBroken?, $\perp$)

recv (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 $(pid, sid, role) \in$ corruptedIntParties:
        reply (corruptInt, true)
    else:
        reply (corruptInt, false)
```

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

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 III 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 B).

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 III-A 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 II-D and II-G).

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 $p =$ `unforgeability` (for, say, $id =$ `public`, modeling public unforgeability), then *<original behavior>* is a check

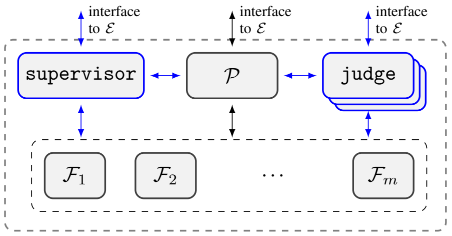


Fig. 4. Example of machines and connections in an accountable real protocol $\mathcal{P}$ with ideal (possibly accountable) subroutines $\mathcal{F}_1, \ldots, \mathcal{F}_m$. Blue components are added by AUC.

during signature validation 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 $\mathcal{F}_{\mathrm{sig}}$ 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* $(\mathtt{privacy}, pid_i)$ *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 $\mathcal{F}^{\mathrm{acc}} := \mathcal{T}_2(\mathcal{T}_1(\mathcal{F}))$ are indeed accountable transformations of $\mathcal{F}$ as per Definition 1.

D. 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 $\mathcal{F}_{\mathrm{sig}}$, 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 III-A. *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 III-B and III-C. *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.

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 II-G).

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 $p \in \mathit{Sec}^{\text{assumption}}$ that relies on an *honest majority* assumption and thus affects everyone (i.e., $id = \mathtt{public}$), e. g., consistency in Bitcoin [46]. 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 II-G, 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., [54], [13]). Our case studies in Sections III-A and III-B provide concrete examples and indeed require a supervisor.

E. Composable Security Analysis in AUC

A security analysis in AUC consists of a realization proof where one shows that the protocol $\mathcal{P}^{\mathrm{acc}}$ at hand indeed realizes (in the UC sense) the accountable ideal functionality $\mathcal{F}^{\mathrm{acc}}$. Among others, this formally proves that $\mathcal{P}^{\mathrm{acc}}$ enjoys the desired security properties, including accountability properties.

[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.

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 II-G and our case studies in Sections III-A and III-C 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.

F. 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. [5] 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 [89], 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.

G. 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 [65], [61], [76], [60], [66], [31], [41], [40], [42], [43], [51], [56]. There are other (often informal) interpretations of accountability, also outside of the field security (cf. [44]). 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., [26], [24]). 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., [67]). 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. [65] 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., [5], [35], [32], [12], [84], [53], [65], [60], [76], [61]).

There is a division in the existing literature concerning relationships of security properties: On the one hand, prior general frameworks for accountability such as [65], [60], [76], [61] 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, [5], [7], [54], [82], [13] 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 B 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 III-A 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 III-A 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 [48], [82], 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 III-A to III-C 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 III-A and III-C, 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., [68], [16], [17], [8]), 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

[4]We note that, as observed by [68], [16], [8], the same UC security proof of the subroutine already implies composition with and without globally available subroutine judges.

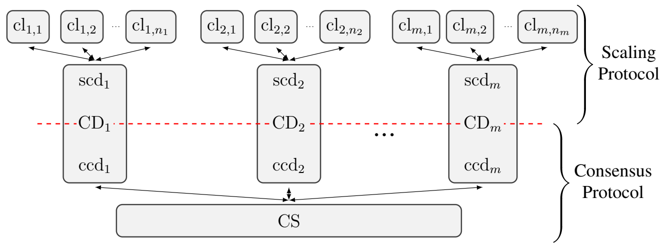


Fig. 5. Consensus scaling: a central service CS establishes consensus. Consensus is distributed via distributors $\text{CD}_1, \ldots, \text{CD}_m$. The CD connect to CS via their CS-clients $\text{ccd}_1, \ldots, \text{ccd}_m$. The CDs distribute consensus via server components $\text{scd}_1, \ldots,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

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 B) 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 III-B) 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.

III. 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. In our technical report [49], we provide for each case study a brief deterrence analysis. Additionally – as a sanity check – we cast existing accountability definitions from the UC MPC literature into AUC in Appendix B.

A. *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], [48], the Hashgraph consensus service [10], and content delivery networks [37].

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 and proofs, are available in our technical report [49].

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 $\text{cl}_{1,1}$, 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 $\text{cl}_{1,1}$, 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. $\text{cl}_{1,1}$ 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):

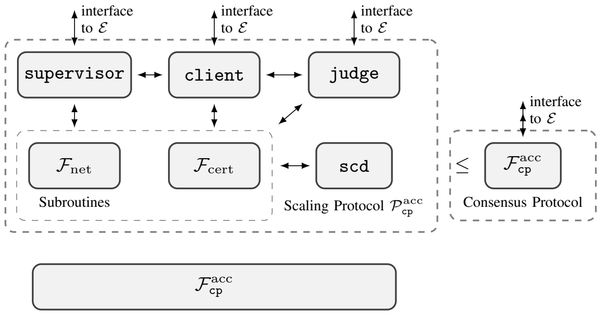
Public individual accountability w.r.t. consistency: Clients (both of the scaling protocol and the CS) obtain a prefix of the same global state or can identify an individual misbehaving party. This definition follows the game-based accountability property formalized for Hyperledger Fabric in [48] but 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., [46], [48], [77], [75], [86]). 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., [50], [9], 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 $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ that captures the above security properties. More specifically, we set $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{consistency}\}$ and $\mathsf{Sec}^{\mathsf{assumption}} = \{\texttt{liveness}\}$. Note that $\texttt{liveness} \notin \mathsf{Sec}^{\mathsf{acc}}$, 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., $\mathsf{pids}_{\mathsf{judge}} = \{\texttt{public}\}$, and considers only assumptions that affect all parties (in a session), i.e., $\mathsf{ids}_{\mathsf{assumption}} = \{\texttt{public}\}$. *(ii)* As soon as liveness is marked broken (for $id = \texttt{public}$), $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ 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 verdict matching the customization in $\mathcal{F}_{\mathrm{judgeParams}}^{\mathrm{acc\text{-}cp}}$. The subroutine $\mathcal{F}_{\mathrm{judgeParams}}^{\mathrm{acc\text{-}cp}}$ 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 $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$, $\mathcal{F}_{\mathrm{judgeParams}}^{\mathrm{acc\text{-}cp}}$'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.



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

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

By this construction, $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ 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 $\texttt{client}$ machine models the code run by the clients $\mathrm{cl}_{i,j}$ and the internal machine $\texttt{scd}$ models code of the SCD. The ideal subroutine $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ 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 $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ (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 $\texttt{client}$ and $\texttt{scd}$, each modeling one party in one protocol session. These parties additionally have access to an ideal signature functionality $\mathcal{F}_{\mathrm{cert}}$ (which includes a PKI) and an ideal network functionality $\mathcal{F}_{\mathrm{net}}$ with bounded message delay δ'. The adversary is allowed to break the assumption of a bounded network delay in $\mathcal{F}_{\mathrm{net}}$ 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 $\texttt{scd}$ and the client component in $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$, but should be considered corrupted as soon as just one of those machines misbehaves. We capture this expected property by considering a party running $\texttt{scd}$ to be corrupted also if the corresponding client in $\mathcal{F}_{\mathsf{cp}}^{\mathrm{acc}}$ is corrupted, i. e., we use the corruption state of $\texttt{scd}$ 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 II-D 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 $\mathcal{F}_{\text{cp}}^{\text{acc}}$ 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 $\mathcal{P}_{\text{cp}}^{\text{acc}}$ using an ideal accountable consensus service subroutine $\mathcal{F}_{\text{cp}}^{\text{acc}}$ is still an accountable consensus service, i. e., all security guarantees are retained and hence the scaled protocol is also a realization of $\mathcal{F}_{\text{cp}}^{\text{acc}}$:

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

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

We provide the formal proof for Theorem 1 in our technical report [49].

Discussion. Using the iUC composition theorem, the ideal subroutine $\mathcal{F}_{\text{cp}}^{\text{acc}}$ of $\mathcal{P}_{\text{cp}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{cp}}^{\text{acc}}$ via another copy of $\mathcal{P}_{\text{cp}}^{\text{acc}}$, 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 $\mathcal{F}_{\text{cp}}^{\text{acc}}$ 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).

B. 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) [30], [69]. 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* [69], 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 III-C, 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 III-A, 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 our technical report [49].

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. Here we consider clients that certify at most one key.[5] More specifically, to certify a new key a client sends a registration request,

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

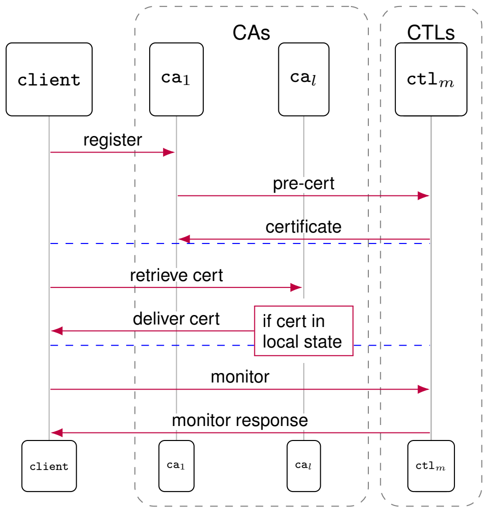


Fig. 7. CTL-based PKI protocol

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 $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$. We formalize the property just sketched starting with a non-accountable ideal PKI functionality $\mathcal{F}_{\mathrm{PKI}}$ that is analogous to Canetti et al.'s ideal functionality $\mathcal{G}_{\mathrm{BB}}$ [26] and the ideal CA $\mathcal{F}_{\mathrm{PKI}}$ from [16]. We then apply AUC to obtain an accountable version $\mathcal{F}_{\mathrm{PKI}}^{\mathrm{acc}}$ that captures accountability w.r.t. certificate correctness.

Clients are the main roles of $\mathcal{F}_{\mathrm{PKI}}$; CTLs and CAs are internal parties of potential realizations of $\mathcal{F}_{\mathrm{PKI}}$. $\mathcal{F}_{\mathrm{PKI}}$ 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, $\mathcal{F}_{\mathrm{PKI}}$ 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 $\mathcal{F}_{\mathrm{PKI}}$ for a certificate of an honest pid' issued by a certain CA, $\mathcal{A}$ is free to choose when and whether $\mathcal{F}_{\mathrm{PKI}}$ answers the request. If $\mathcal{A}$ instructs $\mathcal{F}_{\mathrm{PKI}}$ to respond, $\mathcal{F}_{\mathrm{PKI}}$ provides the unique certificate for pid' (as stored in $\mathcal{F}_{\mathrm{PKI}}$'s state) or it outputs $\perp$ if there is no certificate recorded at that CA. For corrupted clients pid', $\mathcal{F}_{\mathrm{PKI}}$ does not provide any guarantees but lets $\mathcal{A}$ freely determine the response.

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) `ca` role. If `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.

[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.

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 $\mathcal{F}_{\text{psync-net}}$, 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 $\mathcal{P}_{\text{PKI}}^{\text{acc}}$. 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 $\mathcal{P}_{\text{PKI}}^{\text{acc}}$ 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 $\mathcal{P}_{\text{PKI}}^{\text{acc}}$ be the real PKI protocol and $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ be the ideal accountable PKI functionality as described above. Then,*
$$\mathcal{P}_{\text{PKI}}^{\text{acc}} \leq \mathcal{F}_{\text{PKI}}^{\text{acc}}.$$

We provide a formal proof of Theorem 2 in our technical report [49].

C. 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 standard [55] (see Figure 8). UC security of this protocol has already been studied in various settings [25], [18], [19], [64], [16], [26] 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 $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ which can then be realized by $\mathcal{P}_{\text{PKI}}^{\text{acc}}$ (Section III-B) 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 III-B. This also shows that even protocols which traditionally consider only preventive security, such as key exchanges, can benefit from AUC. Due to space constraints, here we explain only the main aspects of this case study; full details, including formal specifications and proofs, are provided in [49].

[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 III-C.

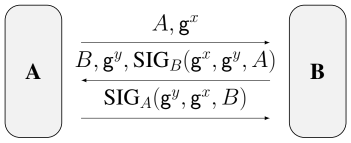

A
B
A, g^x
$B, g^y, \mathsf{SIG}_B(g^x, g^y, A)$
$\mathsf{SIG}_A(g^y, g^x, B)$

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

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 $\mathcal{F}_{\text{KE}}$ [16], [19] and apply AUC to obtain an accountable version $\mathcal{F}_{\text{KE}}^{\text{acc}}$. In $\mathcal{F}_{\text{KE}}^{\text{acc}}$ we consider the local accountability property `authenticity` $\in \mathsf{Sec}^{\mathsf{acc}}$ 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 `authenticity` is marked as broken for Alice by the adversary $\mathcal{A}$ (in exchange for a verdict), then $\mathcal{F}_{\text{KE}}^{\text{acc}}$ 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. $\mathcal{P}_{\text{iso}}^{\text{acc}}$ is a straightforward model of the ISO protocol shown in Figure 8 derived from [16] consisting of the KE protocol part $\mathcal{P}_{\text{pke}}^{\text{acc}}$ and the ideal subroutine $\mathcal{F}_{\text{PKI}}^{\text{acc}}$ which is used for public key distribution. As required by AUC, $\mathcal{P}_{\text{iso}}^{\text{acc}}$ contains local judges and a supervisor. As in Section III-B, the supervisor only forwards the corruption status of internal parties including those that are part of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$, namely $\mathcal{F}_{\text{PKI}}^{\text{acc}}$'s clients, CAs and CTLs. The main idea of the local judge of Alice is to request the verdict of Bobs local judge in $\mathcal{F}_{\text{PKI}}^{\text{acc}}$. If this judge does not complain, then, by definition of $\mathcal{F}_{\text{PKI}}^{\text{acc}}$, 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 $\mathcal{P}^{\mathrm{acc}}_{\mathrm{KE}}$, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{PKI}}$, and $\mathcal{F}^{\mathrm{acc}}_{\mathrm{KE}}$ be as described above and assume that the DDH assumption holds true. Then,*

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

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

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

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

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

IV. Related Work and Conclusion

As already discussed in Section II-G, 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., [12], [32], [71], [54], [13], [23], [33], [78]. As discussed in the introduction and in Appendix B, 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. [13]. 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., [7], [5], [53], [93], [35], [4], [32], [83]. 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 [7], [5] 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 B). 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., [65], [44], [42], [41], [40], [43], [60], [76], [61]. Some of these works are closely related to AUC in that they also consider highly general frameworks for accountability, e.g., [65], [60], [40], [43]. 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* [60]. 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.

V. 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[28] C. Chang and Y. Chang, "Efficient anonymous auction protocols with freewheeling bids," *Comput. Secur.*, vol. 22, no. 8, pp. 728–734, 2003.

[29] 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.

[30] 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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Ethereum Foundation, "Ethereum enterprise," https://www.ethereum.org/enterprise/, 2019, (Accessed on 11/13/2019).

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] ——, "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.

[44] ——, "Accountability in Computing: Concepts and Mechanisms," *Found. Trends Priv. Secur.*, vol. 2, no. 4, pp. 247–399, 2020.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] M. Graf, R. Küsters, and D. Rausch, "AUC: Accountable Universal Composability," Cryptology ePrint Archive, Tech. Rep. 1606, 2022.

[50] 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.

[51] 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.

[52] D. Hofheinz and V. Shoup, "GNUC: A New Universal Composability Framework," *J. Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.

[53] 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.

[54] 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.

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

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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 [68] for a full and revised version.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] ——, "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.

[68] 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.

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

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

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

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

[80] 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.

[81] 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.

[82] M. Rivinius, P. Reisert, D. Rausch, and R. Küsters, "Publicly accountable robust multi-party computation," in *S&P '22*. IEEE, 2022.

[83] 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.

[84] P. Scholl, M. Simkin, and L. Siniscalchi, "Multiparty Computation with Covert Security and Public Verifiability," Cryptology ePrint Archive, Tech. Rep. 2021/366, 2021.

[85] 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.

[86] C. Stathakopoulou, T. David, and M. Vukolic, "Mir-BFT: High-Throughput BFT for Blockchains," *CoRR*, vol. abs/1906.05552, 2019.

[87] 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.

[88] 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).

[89] 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).

[90] 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).

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] ——, "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.

[97] ——, "Evidence and non-repudiation," *Journal of Network and Computer Applications*, vol. 20, no. 3, pp. 267–281, 1997.

APPENDIX

A. 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. [16]. 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** $\langle msg \rangle$ **from** $\langle sender \rangle$ **to** $\langle receiver \rangle$ **s.t.** $\langle$condition$\rangle$:$\langle$code$\rangle$ where $\langle msg \rangle$ is an input pattern, $\langle sender \rangle$ is the receiving interface (I/O or NET), $\langle receiver \rangle$ is the dedicated receiver of the message and $\langle$condition$\rangle$ is a condition on the input. $\langle$code$\rangle$ 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, $\langle msg \rangle$ 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 $\langle$condition$\rangle$ of that block, then uninitialized local variables are initialized with the part of the message that they matched to and $\langle$code$\rangle$ is executed in the context of $\langle receiver \rangle$; no other blocks are executed in this case. If m does not match p or $\langle$condition$\rangle$ 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.** $\langle condition \rangle$ 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. [16]:

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

We use the following additional nomenclature from [16]:

- $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ 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.

B. 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., [12], [32], [71], [54], [13], [23], [33], [78]). These properties include *(publicly) identifiable abort* [54], [13], [32]), *(public/universal) verifiability* [33], [81], [83], *auditability* [12], *openability* [32], and *privacy* [7]. 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. As presented in our technical report [49], the other properties can be dealt with analogously.

Identifiable abort. The standard definition of a basic ideal MPC functionality $\mathcal{F}_{\mathrm{MPC}}$ (cf., e.g., [5], [20], [7], [23]) 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* [5], [7], [54], [82], [13], 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 $\mathcal{F}_{\mathrm{MPC}}^{\text{id-ab}}$ by letting the simulator, during the final output phase, decide whether all honest parties obtain their correct output or provide the name pid of a malicious party that caused the protocol to abort (cf., [54], [32], [13]). Depending on this choice, $\mathcal{F}_{\mathrm{MPC}}^{\text{id-ab}}$ either returns the outputs of f or a special message $(\mathtt{abort}, pid)$ to the honest parties.

Capturing identifiable abort using AUC. We can easily capture the same properties as $\mathcal{F}_{\mathrm{MPC}}^{\text{id-ab}}$ by applying AUC to the basic functionality $\mathcal{F}_{\mathrm{MPC}}$. We set $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{correctness}\}$ and, as part of the transformation $\mathcal{T}_2$, redefine the behavior of $\mathcal{F}_{\mathrm{MPC}}$ 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 $\mathcal{F}_{\mathrm{judgeParams}}$ 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 $\mathcal{F}_{\mathrm{MPC}}^{\mathrm{acc}}$ models the exact same properties as $\mathcal{F}_{\mathrm{MPC}}^{\text{id-ab}}$, with the only syntactical difference being that $\mathcal{F}_{\mathrm{MPC}}^{\text{id-ab}}$ includes the verdict as part of the protocol output while in $\mathcal{F}_{\mathrm{MPC}}^{\mathrm{acc}}$ the verdict is obtained separately from a judge.

We provide a more detailed discussion of MPC via AUC in our technical report [49].