Automated Verification of Accountability in Security Protocols

Robert Künnemann, Ilkan Esiyok and Michael Backes
CISPA Helmholtz Center for Information Security
Saarland Informatics Campus

***Abstract*—Accountability is a recent paradigm in security protocol design which aims to eliminate traditional trust assumptions on parties and hold them accountable for their misbehavior. It is meant to establish trust in the first place and to recognize and react if this trust is violated. In this work, we discuss a protocol-agnostic definition of accountability: a protocol provides accountability (w.r.t. some security property) if it can identify all misbehaving parties, where misbehavior is defined as a deviation from the protocol that causes a security violation.**

We provide a mechanized method for the verification of accountability and demonstrate its use for verification and attack finding on various examples from the accountability and causality literature, including Certificate Transparency and Kroll's Accountable Algorithms protocol. We reach a high degree of automation by expressing accountability in terms of a set of trace properties and show their soundness and completeness.

I. Introduction

The security of many cryptographic protocols relies on trust in some third parties. Accountability protocols seek to establish and ensure this trust by deterrence, often by detecting participants that behave dishonestly. Providing accountability can thus strengthen existing properties in case formerly trusted parties deviate, e.g, the tallying party in an electronic voting protocol or the PKI in a key-exchange protocol. Examples of protocols where accountability is relevant include Certificate Transparency [35] to achieve accountability for the PKI, OCSP stapling [33] to reach accountability for digital certificate status requests in PKI and Accountable Algorithms [21], a proposal for accountable computations performed by authorities.

We regard accountability as a meta-property: given some traditional security property φ, a protocol that provides accountability for φ permits a specific party or the public to determine whether a violation of φ occurred, and if that is the case, which party or parties should be held accountable. Accountability provides an incentive for 'trusted' parties to remain honest, and allows other parties to react to possible violations, e.g., by revocation or fault recovery. It can also be used to build deterrence mechanisms.

For a long time, accountability in the security setting lacked a protocol-independent definition. Generalized definitions and algorithms have been proposed for distributed systems [17], where a complete view of every component is available. In the security setting, however, the problem of identifying dishonest parties is much harder, as they might deviate from the protocol in an invisible manner. In unpublished work, Künnemann *et al.* [24] approach this problem using causal reasoning. Instead of identifying all participants that, perhaps invisibly, deviate from their specifications; they identify the parties that *cause* a violation. Even if a limited part of the communication is available, cryptographic mechanisms such as digital signatures, commitments and zero-knowledge proofs can be used to leave traces that can be causally related to the violation, e.g., a transmission of a secret.

In this paper, we provide the first mechanized verification technique for their approach, which was stated in a custom process calculus in which several parties can choose individual ways of misbehaving. First, we propose a variant of their definition in what we call the *single-adversary* setting. In this setting, a single adversary controls all dishonest parties. This setting is used in almost all existing protocol verification tools. This vastly simplifies the definition of accountability and enables the use of off-the-shelf protocol verifiers. Second, we give verification conditions implying accountability for a specific *counterfactual relation*. This relation links *what actually happened* to *what could have happened*, e.g., if only a subset of parties had mounted an attack. Causation and, as we will demonstrate, accountability depend on how this relation is specified. We use these verification conditions and off-the-shelf tools to automatically verify Certificate Transparency, OCSP stapling and other protocols for accountability w.r.t. this relation. However, more complex scenarios require specifying more fine-grained counterfactual relations. We show for the general case: this definition can be decomposed into several trace properties and the decomposition is sound and complete. For our case studies, the verification conditions consist of 7 to 31 such properties; most of them can be verified within tenths of seconds. Third, we implement this verification method for an extension of the applied-π calculus to provide a convenient toolchain for the specification and verification of accountability protocols. We verify accountability or find attacks on *a*) several toy examples that highlight the complexity of accountability, *b*) an abstract modelling of the case where accountability is achieved by maintaining a central audit log, *c*) several examples from the causality literature, *d*) OCSP-Stapling, *e*) Certificate Transparency and *f*) *Accountable Algorithms*, a seminal protocol for accountable computations in a real-world setting. We thus list our contributions as follows:

- a new definition of accountability in the single-adversary setting (which is simpler than the previous definition [24]

due to the single-adversary setting).
- a verification technique, based on a sound and complete translation to a set of trace properties that is compatible with a mature and stable toolchain for protocol verification.
- a demonstration on several case studies, including two fixes to Kroll's Accountable Algorithms protocol, and a machine-verified proof that both fixes entail accountability for both participants.

II. Accountability

To define accountability, we follow the intuition that a protocol provides accountability if it can always determine which parties caused a security violation. In this sense, accountability is a meta-property: we speak of accountability for violations of φ. If no violation occurs, no party causes it; if no party causes a violation, there is no violation. Hence accountability w.r.t. φ implies verifiability of φ [27].

To reason about failures, i.e., violations of φ, our formalism has to allow parties to deviate from the protocol. Each party is either *honest*, i.e., it follows the behaviour prescribed by the protocol, or it is *dishonest*, i.e., it may *deviate* from this behaviour. Often there is a judging party, which is typically *trusted*, i.e., it is always honest. A dishonest party is not necessarily deviating, it might run a process that is indistinguishable from its protocol behaviour. Hence it is impossible to detect all dishonest parties.

In the security setting, parties cannot monitor each other completely. Typically, they only receive messages addressed or redirected to them. This includes the judging party. Therefore, a deviating party A can send a secret to another deviating party B and the judging party may not notice. Under these circumstances, identifying all deviating parties is also impossible.

Instead, we focus on identifying all parties *causing* the violation of the security property φ. The protocols we consider in this work are designed in a way the parties that are deviating (and thus dishonest) will have to leave evidence in order to cause security violations.

The definition we provide here is a simplified version of an earlier, causality-based definition [24], adapted to a setting where there is only a single adversary controlling all deviating parties, or equivalently, where the deviating parties share all the knowledge they obtain. Both definitions are based on sufficient causation [12, 22]. The intuition is to capture all parties for which the *fact that they are deviating at all* is causing the violation. This may not only be a single party, but also a set of parties. If two parties A, B collude against a secret sharing scheme with a threshold of two and expose some secret, the single party A would not be considered a cause, but we would say that $\{A, B\}$ have *jointly* caused the failure.

Assume a fixed finite set of parties $\mathcal{A} = \{A, B, \ldots\}$. Intuitively, the fact that a party or a set of parties $\mathrm{S} \subseteq \mathcal{A}$ are deviating is a cause for a violation iff:

SC1. A violation indeed occurred and S indeed deviated.

SC2. If all deviating parties, *except the parties in* S, behaved honestly, the same violation would still occur.

SC3. S is minimal, i.e., SC1 and SC2 hold for no strict subset of S.

The first condition is self-explanatory. The second formalizes that the misbehaviour of the parties in S alone is *sufficient* to disrupt the protocol. For the secret sharing scheme scenario above, if the party A deviated alone, this would not cause the exposure of the secret, so SC2 would not hold. $\{A, B\}$, however, would meet all of the conditions. SC2 reasons about a scenario that is different from the events that actually took place, which is called a *counterfactual* in causal reasoning. At the end of this section, we will discuss different *counterfactual relations* between the actual and the counterfactual course of events. The third condition SC3 ensures minimality. It removes parties that deviated, but whose deviations are not relevant to the coming about of the violation.

Note that there can be more than one (joint) cause. If the aforementioned protocol would run in two sessions, one in which A and B colluded to expose the secret and another one in which A and C colluded to expose a different secret, there would be two causes, $\{A, B\}$ and $\{A, C\}$, each individually satisfying SC1 to SC3, each being an individual cause for the failure. This nicely separates joint causes (A and B working together; A and C working together) from independent causes (the first collusion and the second collusion).

Intuitively, an accountability mechanism provides accountability for φ if it can always point out all $\mathrm{S} \subseteq \mathcal{A}$ causing a violation of φ, or $\varnothing$ if there is no violation.

Need for trusted parties: Accountability protocols aim at eliminating trust assumptions, yet they often require a trusted judging party. But, if the judging party operates offline and their judgement can be computed with public knowledge, then any member of the public can replicate the judgement and validate the result. The judging party is thus not a third party, but the observer itself, who we assume can be trusted to follow the protocol, i.e., not cheat herself.

Individually deviating parties vs. single adversary: Protocol verification tools usually consider a single adversary representing both an intruder in the network and untrusted parties. Typically, the attacker sends a *corruption* message to these parties, they accordingly transmit their secrets to the adversary. As messages are transmitted over an insecure network, the adversary can then impersonate the parties with their secrets. This is a sound over-approximation in case of security properties like secrecy and authentication, as the adversaries are stronger if they cooperate. It also simplifies verification: a single adversary can be modelled in terms of non-deterministic state transitions for message reception (see, e.g., [2, Theorem 1]). Many protocol verification tools operate in this way [36, 9, 11, 14]. We therefore focus on the single-adversary setting, in order to exploit existing methods for protocol verification and encourage the development of accountability protocols. This is the main distinction between our simplified definition of accountability and the earlier definition [24]. Nevertheless, the philosophical difference be-

tween these two settings remains relevant for our analysis, as it is based on causal reasoning about deviating parties. In Section IX, we discuss the difference between these two settings and elaborate on our current understanding of them.

Protocol accountability: Let a protocol P be defined in some calculus, e.g., the applied-π calculus, and assume that we can define the set of traces induced by the protocol, denoted $traces(P)$. We also assume a *counterfactual relation* r between an actual trace and a counterfactual trace. Depending on the protocol, different counterfactual relations can capture the desired granularity of a judgement. We will describe this relation and give examples in the following section.

Given a trace t, a security property φ can be evaluated, e.g., $t \vDash \varphi$ iff φ is true for t, which we will sometimes abbreviate to $\varphi(t)$. We assume that any trace $t \in traces(P)$ determines the set of dishonest agents, i.e., those who received the corruption message from the adversary and define $corrupted(t) \subseteq \mathcal{A}$ to be this set.

We can now define the a posteriori verdict (apv), which specifies all subsets of $\mathcal{A}$ that are sufficient to cause $\neg\varphi$. The task of an accountability protocol is to always compute the apv, but without having full knowledge of what actually happened, i.e., t.

Definition 1 (a posteriori verdict)**.** *Given an accountability protocol P, a property φ, a trace t and a relation r, the a posteriori verdict, which formalizes the set of parties causing $\neg\varphi$, is defined as:*

$$apv_{P,\varphi,r}(t) := \{\mathrm{S} \mid t \vDash \neg\varphi \text{ and } \mathrm{S} \text{ minimal s.t.} \\ \exists t' : r(t,t') \wedge corrupted(t') = \mathrm{S} \wedge t' \vDash \neg\varphi\}.$$

Relation r is reflexive, transitive and $r(t,t')$ implies $corrupted(P,t') \subseteq corrupted(P,t)$.

Each set of parties $\mathrm{S} \in apv_{P,\varphi,r}(t)$ is a sufficient cause for an eventual violation, in the sense outlined at the beginning of this section. The condition $t \vDash \neg\varphi$ ensures that indeed a violation took place. If the parties in S did not deviate in t, then S would not be minimal, hence SC1 holds. SC3 holds by the minimality condition. The remaining conditions capture SC2: t' is a *counterfactual trace* w.r.t. t, a trace contrary to what actually happened.[1] In t', only the parties in S may deviate, which should suffice to derive a violation. We define the relation r to constrain the set of counterfactual traces. At the very least, the condition that $corrupted(t') \subseteq corrupted(t)$ should hold to guarantee that for any violating trace, a minimal S is defined.[2] We will discuss a few variants of r in the following section.

For a given trace, the apv outputs a set of sufficient causes, i.e., a set of sets of agents. We call this output as the *verdict*. We also remark that $apv_{P,\varphi,r}(t) = \varnothing$ iff $t \vDash \varphi$, i.e., an empty verdict means the absence of a violation — there can be no cause for an event that did not happen.

To abstract from the mechanism by which an accountability protocol announces the purported set of causes for a violation — this could range from a designated party computing them to a public ledger that allows the public to compute it — we introduce a *verdict function* from $traces(P) \to 2^{2^{\mathcal{A}}}$. An accountability protocol is thus described by P and the verdict function. We can now state accountability: a verdict function provides accountability if it always computes the apv.

Definition 2 (accountability)**.** *A verdict function $verdict : traces(P) \to 2^{2^{\mathcal{A}}}$ provides a protocol P with accountability for a property φ (w.r.t. r) if, for any trace $t \in traces(P)$*

$$verdict(t) = apv_{P,\varphi,r}(t).$$

Example 1. *Assume a protocol in which a centralized monitor controls access to some resource, e.g., confidential patient records. Rather than preventing exceptional access patterns, e.g., requesting the file of another doctor's patient, the requests are allowed, but logged by the monitor. Accountability tends to be preferable over prevention in such cases, e.g., in case of emergencies.*

The set of agents comprises doctors D_1, D_2 and the centralized monitor. The centralized monitor is trusted and effectuates requests only if they are digitally signed. The security property φ is true if no exceptional request was effectuated. Per protocol, D_1 and D_2 never send exceptional requests, however, if in trace t, the central adversary corrupts them and learns their signing keys, he can act on their behalf and sign exceptional requests. The set of dishonest parties contains those corrupted by the adversary, $corrupted(t) = \{D_1, D_2\}$. A verdict $\{\{D_1\},\{D_2\}\}$ indicates that both D_1 and D_2 deviated in a way that caused a violation, i.e., an exceptional request was effectuated. If a third doctor D_3 was also in the protocol and it was corrupted $D_3 \in corrupted(t)$, but never signed any request, then it would not be involved in the apv $apv_{P,\varphi,r}(t) = \{\{D_1\},\{D_2\}\}$. The apv for this protocol can be computed by only taking the log into account: a verdict function $verdict$, that operates on the monitor's log, can be defined to compose singleton verdicts for each party that signed an exceptional request. This verdict function is easy to implement, yet it provides the protocol with accountability for φ because it computes $apv_{P,\varphi,r}$ correctly.

Counterfactual relation: The relation r in the definition of the apv defines which counterfactual scenarios are deemed relevant in SC2. While there is an agreement in the causality literature that t' cannot be chosen arbitrarily, there is no agreement on *how* they should relate. We slightly change the previous monitoring example and discuss why the relation is important.

Example 2. *Assume that the monitor supports a second mechanism to handle requests. Here, a doctor D_1 can also sign his exceptional request and ask his chief of medicine*

[1] Technically, the set of counterfactual traces includes $t' = t$, as r is reflexive, and thus not every instance of t' is, strictly speaking, a counterfactual. For brevity, we prefer to nevertheless call t' a counterfactual, despite this imprecision.

[2] Consider r total, t with $t \vDash \neg\varphi$ and $corrupted(t) = \{A\}$ and t' with $t' \vDash \neg\varphi$ and $corrupted(t') = \{B\}$ as a counterexample. Neither $\{A\}$ nor $\{B\}$ would be minimal.

C to approve it. Assume in trace t, both D_1 and C collude and use this mechanism to effectuate an exceptional request, violating φ. Intuitively, one would expect the apv, relying on logs, to give the verdict $\{\{D_1, C\}\}$. However, D_1 could have used the first *mechanism for this request. Hence, there is a counterfactual trace t' where only D_1 is dishonest. If $r(t, t')$, then the more intuitive verdict $\{\{D_1, C\}\}$ is not minimal, but $apv_{P,\varphi,r}(t) = \{\{D_1\}\}$ is minimal, shifting the blame to D_1 alone. The intuitive response would be: 'But that is not what happened!', which is precisely what r needs to capture.*

We discuss three approaches for relating factual and counterfactual traces:

- *by control-flow:* $r_c(t, t')$ iff t and t' have similar control-flow. Several works in the causality literature relate traces by their control-flow [12, 24, 26], requiring counterfactuals to retain, to varying degree, the same control-flow. See [25] for a detailed discussion about control flow in Pearl's influential causation framework. For simplicity, the notion we present (cf. Section VII) captures only the control-flow of trusted parties, i.e., parties guaranteed to be never controlled by the adversary. In case of the example, the control-flow of the trusted monitor would distinguish these two mechanisms.
- *by kind of violation:* $r_k(t, t')$ iff t and t' describe the same kind of violations. This approach is, e.g., used in criminal law to solve causal problems where the classical 'what-if' test fails, e.g., a person was poisoned, but shot before the poison took effect. The classical test for causality (sine qua non) gives unsatisfying results (without the shot, the person would still have died), unless one describes the causal event in more detail, i.e., by distinguishing death from shooting from death from poisoning ([13, p. 188]; see also [31, p. 46]). For security protocols, the instance of the violation could be characterized by the session, the participating parties or other variables that are free in in the security property φ (see Lowe [30]). This relation is informal and depends on intuition, so it is not used in our analysis.
- *weakest relation according to Def. 1:* $r_w(t, t') \iff corrupted(P, t') \subseteq corrupted(P, t)$. This relation is conceptually simple and suitable for many protocols in which collusion is not an issue, i. e., verdicts contain only singleton sets. Outside this class, it may give unintuitive a posteriori verdicts in cases where t requires collusion, but one of the colluding parties could mount a possibly very different attack by themselves.

In Section III, we provide verification conditions for r_w. In Section IV, we provide more general verification conditions that apply to arbitrary relations, as some scenarios require more fine-grained analysis, and later we mechanize it for r_c.

III. Verification conditions for r_w

In this section, we define a set of verification conditions parametric in a security property φ and a verdict function. If these conditions are met, they provide a protocol with accountability for φ w.r.t. the weakest condition on counterfactuals $r_w(t, t') := corrupted(t') \subseteq corrupted(t)$. Each of these conditions is a trace property and can thus be verified by off-the-shelf protocol verification tools. In our case studies, we will use these verification conditions to verify accountability properties for the Certificate Transparency protocol.

TABLE I
Verification Conditions *for* rw.

conditions	formulae
Exhaustiveness (XH)*:*	$\forall t.\quad \omega_1(t) \vee \cdots \vee \omega_n(t)$
Exclusiveness (XC)*:*	$\forall t, i, j.\quad i \neq j \implies \neg(\omega_i(t) \wedge \omega_j(t))$
Sufficiency of each ω_i *s.t.* $V_i \neq \varnothing$ ($\mathrm{SF}_{\omega_i,\varphi,\mathrm{S}}$)*:*	$\forall \mathrm{S} \in V_i.\quad \exists t.$ $\neg\varphi(t) \wedge corrupted(t) = \mathrm{S}$
Verifiability of each ω_i ($\mathrm{V}_{\omega_i,V_i}$)*:*	$\forall t.\quad \omega_i(t) \implies$ $(V_i = \varnothing \iff \varphi(t))$
Minimality of each V_i (M_{φ,V_i})*:*	$\forall \mathrm{S} \in V_i\ \forall \mathrm{S}' \subsetneq \mathrm{S}$ $\nexists t.\ \neg\varphi(t) \wedge corrupted(t) = \mathrm{S}'$
Uniqueness of each V_i ($\mathrm{U}_{\omega_i,V_i}$)*:*	$\forall t.\quad \omega_i(t) \implies$ $\bigcup_{\mathrm{S}\in V_i} \mathrm{S} \subseteq corrupted(t)$
Completeness of each V_i (C_{V_i})*:*	$\forall \mathrm{S} \subseteq \bigcup_{\mathrm{S}'\in V_i} \mathrm{S}'\ \forall j.\ V_j = \{\mathrm{S}\} \implies$ $\mathrm{S} \in V_i.$

(t is quantified over $traces(P)$)

The main idea: we assume the verdict function is described as a case distinction over a set of trace properties ω_1 to ω_n. Any of these *observations* ω_i is then assigned a verdict V_i.

Definition 3 (verification conditions)**.** *Let $verdict$ be a verdict function of form:*

$$verdict(t) = \begin{cases} V_1 & \text{if } \omega_1(t) \\ \vdots & \\ V_n & \text{if } \omega_n(t) \end{cases}$$

and φ a predicate on traces. We define the verification condition $\gamma_{\varphi,verdict}$ as the conjunction of the formulae in Table I.

We briefly go over these conditions. The case distinction needs to be exhaustive (XH) and exclusive (XC), because verdict functions are total. For any observation ω_i that leads to a non-empty verdict, any set of parties S in this verdict needs to be able to produce a violating trace on their own ($\mathrm{SF}_{\omega_i,\varphi,\mathrm{S}}$). However, removing any element from S should make it impossible to produce a violation (M_{φ,V_i}), due to the minimality requirement in Def. 1. If an observation leads to the empty verdict, it needs to imply the security property φ, because accountability implies verifiability ($\mathrm{V}_{\omega_i,V_i}$). Whenever an observation ω_i is made, all parties that appear in the ensuing verdict have necessarily been corrupted ($\mathrm{U}_{\omega_i,V_i}$). This ensures uniqueness; if there was a second sufficient and minimal verdict, part of the verdict would correspond to a trace that corrupts parties that do not appear in the verdict (details in the proof of completeness, Appendix C in the full version [23]). Finally, if there is a singleton verdict (e.g.,

$V_j = \{\{B, C\}\}$) containing only parties that appear in another composite verdict (e.g., $V_i = \{\{A, B\}, \{A, C\}\}$) then traces that give the former are related to traces that give the latter (where, at least, A, B and C were dishonest). Hence the singleton verdict needs to be included. (C_{V_i}).

We show these conditions sound and complete in Appendix C in the full version [23]. Practically, this means that any counter-example to any lemma generated from these conditions demonstrates an attack against accountability.

Example 3. *Consider the centralized monitor from Example 1, and, for simplicity, assume there is only one doctor D. The verdict function gives $V_1 = \{\{D\}\}$ if it logged an action signed by D, if this action was effectuated and if it was exceptional. Otherwise, it gives $V_2 = \varnothing$. To show that his verdict function provides accountability for $\varphi :=$ no exceptional action was effectuated, one would show:*

- *the case distinction exhaustive and exclusive,*
- *that the attacker can effectuate an exceptional action if D's signing key is known ($\mathrm{SF}_{\omega_i,\varphi,\mathrm{S}}$),*
- *that the 'otherwise' condition (no exceptional action was effectuated and signed by D) implies that no exceptional action was effectuated by anyone ($\mathrm{V}_{\omega_i,V_i}$),*
- *that no exceptional action can be effectuated without knowledge of D's signing key (M_{φ,V_i}), and*
- *that D's signature on an exceptional action that was effectuated can only be obtained by corrupting D ($\mathrm{U}_{\omega_i,V_i}$).*
- *Completeness (C_{V_i}): V_1 is the only non-empty verdict.*

IV. Verification conditions for arbitrary r

As outlined in Example 2, there are scenarios where a more fine-grained analysis is necessary. These scenarios are characterized by violations that can be provoked either by a set of colluding parties or by a subset thereof, using a different mechanism. Hence we provide a different and more elaborate set of verification criteria (see Table II). They fall into two categories: the first consists of trace properties that again can be verified using off-the-shelf protocol verifiers. The second relates the case distinction used to define the verdict to the relation: in general, all traces that fall into the same case should be related. The verification of the second kind of conditions depends on the relation chosen and can be conducted by hand. In a later section, we mechanize the verification of these conditions for the relation r_c specifically. Hence our method is fully automated for this relation.

These verification criteria are sound and complete. (see Appendix D in the full version [23]). This means that the conjunction of all verification criteria is logically equivalent to accountability and thus contradicts Datta et. al.'s view that 'accountability depends on actual causation and it is not a trace property' [12].

Definition 4 (verification conditions)**.** *Let $verdict$ be a verdict function of the form from Definition 3 and φ be a predicate on traces. We define the verification condition $\nu_{\varphi,verdict}$ as the conjunction of the formulae in Table II, where t and t_i range over $traces(P)$.*

TABLE II
Verification conditions *for arbitrary r.*

conditions	formulae
Exhaustiveness (XH)*:*	$\forall t.\quad \omega_1(t) \vee \cdots \vee \omega_n(t)$
Exclusiveness (XC)*:*	$\forall t, i, j.\ i \neq j \implies \neg(\omega_i(t) \wedge \omega_j(t))$
Sufficiency for each ω_i, with singleton $V_i = \{\mathrm{S}\}$ ($\mathrm{SFS}_{\omega_i,\varphi,V_i=\{\mathrm{S}\}}$)*:*	$\exists t.\quad \omega_i(t) \wedge \neg\varphi(t)$ $\wedge\, corrupted(t) = \mathrm{S}$
Sufficiency for each ω_i, with $\lvert V_i\rvert \geqslant 2$ ($\mathrm{SFR}_{R,\omega_i,\varphi,\lvert V_i\rvert\geqslant 2}$)*:*	$\forall \mathrm{S}.\quad \mathrm{S} \in V_i \implies$ $\exists j.\quad R_{i,j} \wedge V_j = \{\mathrm{S}\}$
Verifiability for each ω_i, ($\mathrm{V}_{\omega_i,V_i}$)*:*	$\forall t.\quad \omega_i(t) \implies$ $(V_i = \varnothing \iff \varphi(t))$
Minimality, for singleton $V_i = \{\mathrm{S}\}$ (M_{φ,V_i})*:*	$\forall \mathrm{S}'.\quad \mathrm{S}' \subsetneq \mathrm{S} \implies$ $\nexists t.\quad \omega_i(t) \wedge corrupted(t) = \mathrm{S}'$
Minimality, for composite V_i, $\lvert V_i\rvert \geqslant 2$ ($\mathrm{M}_{R,\lvert V_i\rvert\geqslant 2}$)*:*	$\nexists \mathrm{S}, \mathrm{S}' \in V_i \quad \mathrm{S}' \subsetneq \mathrm{S}$
Uniqueness of each V_i with $V_i = \{\mathrm{S}\}$ ($\mathrm{U}_{\omega_i,V_i=\{\mathrm{S}\}}$)*:*	$\forall t.\quad \omega_i(t) \implies \mathrm{S} \subseteq corrupted(t)$
Completeness of $\lvert V_i\rvert \geqslant 2$ ($C_{\lvert V_i\rvert\geqslant 2}$)*:*	$\forall j, S.\quad R_{i,j} \wedge V_j = \{\mathrm{S}\} \implies$ $\exists \mathrm{S}'.\quad \mathrm{S}' \in V_i \wedge \mathrm{S} \subseteq \mathrm{S}'$
Relation is lifting of r ($\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$)*:*	if $V_i, V_j \neq \varnothing$ then $\forall t, t'.\quad \omega_i(t) \wedge \omega_j(t')$ $\wedge R_{i,j} \iff r(t, t')$
Relation is reflexive and terminating on singleton ($\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$)*:*	$\forall i, j.\quad V_i$ is singleton $\wedge R_{i,i} \wedge (R_{i,j} \implies i = j)$

(t is quantified over $traces(P)$)

Again, we assume the verdict to be expressed as a case distinction. This case distinction must be sufficiently fine-grained to capture all relevant classes of counterfactuals, e.g., all ways the violation could come about in terms of r. We distinguish between the empty verdict (meaning no violation took place), singleton verdicts ($\{\mathrm{S}\}$, where S itself is a set of parties jointly causing a violation) and composite verdicts (consisting of two or more elements, e.g., $\{\{A, B\}, \{A, C\}\}$ if A, B and C deviated, but A could have caused the violation either jointly with B or with C). The main idea is that the correctness of composite verdicts, e.g., $\{\{A, B\}, \{C\}\}$, follows from the correctness of the singleton verdicts they are composed from, e.g., $\{\{A, B\}\}$ and $\{\{C\}\}$, as long as all traces that provoke the composite verdict relate to the singleton verdict.

We assume the cases to be connected along these lines; all cases resulting in an empty verdict have to guarantee φ to hold ($\mathrm{V}_{\omega_i,V_i}$). All cases resulting in singleton verdicts have to imply that (a) a violation took place ($\mathrm{V}_{\omega_i,V_i}$), (b) that the parties in the verdict alone can provoke this violation ($\mathrm{SFS}_{\omega_i,\varphi,V_i=\{\mathrm{S}\}}$), (c) these parties need to be corrupted whenever this case matches (M_{φ,V_i}) and (d) that the verdict is unique ($\mathrm{U}_{\omega_i,V_i=\{\mathrm{S}\}}$).

Composite verdicts need to relate to singleton verdicts by

means of a lifting R of the relation r ($\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$). For each part of a composite verdict, R points to the singleton verdicts for the same parties. Therefore, it needs to be terminating on singleton cases ($\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$). As R is a lifting of r and reflexive (i.e., all traces in the same case are related to each other, $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$), completeness, sufficiency and minimality carry over from singleton cases to composite cases, as long as all parts of of the composite verdict are covered by a singleton verdict ($\mathrm{SFR}_{R,\omega_i,\varphi,|V_i|\geqslant 2}$), and, vice versa, all related singleton verdicts are contained in the composite verdicts ($C_{|V_i|\geqslant 2}$). In this way, we can avoid the requirement that the composite verdict itself needs to define the minimal set of parties needed to provoke a violation, which would not be the case in most of our case studies.[3] Instead, we only need a simple syntactic check ($\mathrm{M}_{R,|V_i|\geqslant 2}$) to ensure that the parts of a composite verdict are not contradictory with regards to the minimality of the apv. Consider, e.g., the composite verdict $\{\{A, B\}, A\}$.

In summary, the key to these verification conditions is to express equivalence classes w.r.t. r and relations between them in the case distinction describing the verdict function.

Example 4. *Consider the extended centralized monitor (Example 2) and assume that the logged signature distinguishes which mechanism was used to effectuate an action and that only one action can be effectuated. Assume further a verdict function that (a) outputs $\{\{D\}\}$ if the monitor logged and effectuated an exceptional action signed by D, (b) outputs $\{\{C, D\}\}$ if it logged and effectuated an exceptional action signed by D and C, and (c) outputs $\varnothing$ otherwise. Minimality can only hold if case (a) and case (b) are not in the relation. Hence $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ requires any trace falling into case (a) to be unrelated from any trace falling in case (b). $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$ requires all traces leading to the observation in case (a) to be related, and likewise for case (b). If we consider, e.g., the monitor's control flow r_c, this can be shown automatically (cf. Section VII). If we consider r_k, it is essentially an axiom.*

V. Calculus

Before we present our case studies, we will elaborate on the protocol calculus in which they are stated. The calculus we used is an extension of the well-known applied-π calculus [2]. In addition to the usual operators for concurrency, replication, communication, and name creation, from applied-π, this calculus (called SAPiC [20, 6][4]) supports constructs for accessing and updating an explicit global state, which is useful for accountability protocols that rely on trusted third parties retaining some state or a public ledger. Readers familiar with the applied-π calculus can jump straight to Section V-B, where the modelling of corruption is explained. The constructs for state manipulation are marked in Figure 1.

We will now introduce the syntax and informally explain the semantics of the calculus. For the formal semantics, please refer to Appendix A.

[3]Compare with the minimality requirement in Table I in Def. 3.

[4] Our results apply to both the original version of SAPiC and the extension with reliable channels.

Terms and equational theories: Messages are modelled as abstract terms. We define an order-sorted term algebra with the sort msg and two incomparable subsorts pub and $fresh$ for two countably infinite sets of public names (PN) and fresh names (FN). Furthermore, we assume a countably infinite set of variables for each sort s, $\mathcal{V}_s$. Let $\mathcal{V}$ be the union of the set of variables for all sorts. We write $u : s$ when the name or variable u is of sort s. Let Σ be a signature, i.e., a set of function symbols, each with an arity. We write f/n when function symbol f is of arity n. There is a subset $\Sigma_{priv} \subseteq \Sigma$ of *private* function symbols which cannot be applied by the adversary. Let Terms be the set of well-sorted terms built over Σ, PN, FN and $\mathcal{V}$, and $\mathcal{M}$ be the subset containing only ground terms, i.e., terms without variables.

Equality is defined by means of an equational theory E, i.e., a finite set of equations between terms inducing a binary relation $=_E$ that is closed under application of function symbols, bijective renaming of names and substitution of variables by terms of the same sort.

Example 5. *We model digital signatures using symbols $\{sig, verify, pk, sk, true\} \subset \Sigma$ with $sk \in \Sigma_{priv}$, and equation*

$$verify(sig(m, sk(i)), m, pk(sk(i))) = true.$$

For the remainder of the article, we will assume the signature Σ and equational theory E to contain symbols and equations for pairing and projection $\{\langle .,. \rangle, \mathsf{fst}, \mathsf{snd}\} \subseteq \Sigma$ and equations $\mathsf{fst}(\langle x, y\rangle) = x$ and $\mathsf{snd}(\langle x, y\rangle) = y$ are in E. We use $\langle x_1, x_2, \ldots, x_n\rangle$ as a shortcut for $\langle x_1, \langle x_2, \langle \ldots, \langle x_{n-1}, x_n\rangle \ldots\rangle$. We suppose that functions between terms are interpreted modulo E, i.e., if $x =_E y$ then $f(x) = f(y)$.

Facts: We also assume an unsorted signature Σ_{fact}, disjoint from Σ. The set of *facts* is defined as $\mathcal{F} := \{F(t_1, \ldots, t_k) \mid t_i \in \mathrm{Terms}_\Sigma, F \in \Sigma_{fact} \text{ of arity } k\}$ and used to annotate protocol steps.

Sets, sequences, and multisets: We write $\mathbb{N}_n$ for the set $\{1, \ldots, n\}$. Given a set S, we denote the set of finite sequences of elements from S, by S^*. Set membership modulo E is denoted by $\in_E$ and defined as $e \in_E S$ iff $\exists e' \in S.\ e' =_E e$. $\subset_E$, $\cup_E$, and $=_E$ are defined for sets in a similar way. Application of substitution is lifted to sets, sequences and multisets as expected. By abuse of notation we sometimes interpret sequences as sets or multisets; the applied operators should make the implicit cast clear.

A. Syntax and informal semantics

0 denotes the terminal process. $P \mid Q$ is the parallel execution of processes P and Q and $!P$ the replication of P allowing an unbounded number of sessions in protocol executions. $P + Q$ denotes *external* non-deterministic choice, i.e., if P or Q can reduce to a process P' or Q', $P + Q$ may reduce to either. The construct $\nu a; P$ binds the name $a \in FN$ in P and models the generation of a fresh, random value. The processes $\mathrm{out}(m, n);\ P$ and $\mathrm{in}(m, n);\ P$ represent the output, respectively input, of message n on channel m. As opposed to

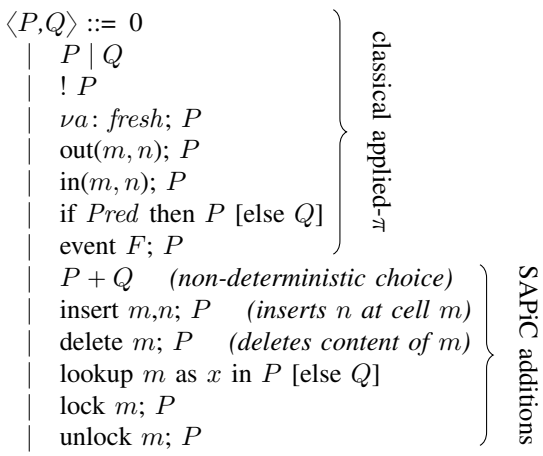

Fig. 1. Syntax ($a \in FN$, $x \in \mathcal{V}$, $m, n \in$ Terms $Pred \in \mathcal{P}$, $F \in \mathcal{F}$).

the applied pi calculus [2], SAPiC's input construct performs pattern matching instead of variable binding. If the channel is left out, the public channel c is assumed, which is the case in the majority of our examples. The process if $Pred$ then P else Q will execute P or Q, depending on whether $Pred$ holds. For example, if $Pred = equal(m, n)$, and $\phi_{equal} = x_1 \approx x_2$, then if $equal(m, n)$ then P else Q will execute P if $m =_E n$ and Q otherwise. (In the following, we will use $m = n$ as a short-hand for $equal(m, n)$). The event construct is merely used for annotating processes and will be useful for stating security properties. For readability, we sometimes omit trailing 0 processes and else branches that consist of a 0 process. The remaining constructs are used to manipulate state and were introduced with SAPiC [20]. The construct insert m,n binds the value n to a key m. Successive inserts overwrite this binding, the delete m operation 'undefines' the binding. The construct lookup m as x in P else Q allows for retrieving the value associated to m, binding it to the variable x in P. If the mapping is undefined for m, the process behaves as Q. The lock and unlock constructs are used to gain or waive exclusive access to a resource m, in the style of Dijkstra's binary semaphores: if a term m has been locked, any subsequent attempt to lock m will be blocked until m has been unlocked. This is essential for writing protocols where parallel processes may read and update a common memory.

Example 6. *The centralized monitor from Example 1 can be modelled as follows. For $NormalAct/0$, we can model the doctor's role as follows:*

D:=in(a); if a = $NormalAct$ then
out($\langle\langle$'Do', $a\rangle$,$sign(\langle$'Do', $a\rangle$,sk('D'))$\rangle$).

The centralized monitor itself verifies the signature and logs the access using the event construct. Note that it does not check whether a constitutes a 'normal' action.

M:= (in($\langle m_1 := \langle$'Do', $a\rangle$,$m1s\rangle$);
if *verify* ($m1s$,m_1,$pk(sk$('D')))=$true()$ then
event $LogD(a)$;event $Execute(a)$)

To model these parties running arbitrarily many sessions in parallel, we compose D and M to $!D \,|!M$.

As usual, the semantics are defined by means of a reduction relation. A configuration c consists of the set of running processes, the global store and more. By reducing some process, it can transition into a configuration c'. This relation is denoted $c \xrightarrow{F} c'$, where the fact F denotes an event, e.g., $LogD(a)$, or the adversary sending a message m, in which case $F = K(m)$.

An *execution* is a sequence of related configurations, i.e., $c_1 \xrightarrow{F_1} \cdots \xrightarrow{F_n} c_{n+1}$ The sequence of non-empty facts F_i defines the trace of an execution. Given a ground process P, $traces(P)$ is the set of traces that start from an initial configuration c_0 (no messages emitted yet, no open locks etc.).

To specify trace properties, SAPiC and the underlying Tamarin prover [36] support a fragment of first-order logic with atoms $F@i$ (fact F is at position i in trace), $i < j$ (position i precedes position j) and equality on positions and terms (see. Appendix B for its formal definition). We write $t \vDash \varphi$ if t satisfies a trace property φ.

B. Accountability protocols

A process by itself does not encode which of its subprocesses represents which agent. Hence, for each agent A, we assign a process P_A. Furthermore, in order to model the adversary taking control of agents, each agent needs to specify a corruption procedure. At the least, this corruption procedure outputs that agent's secrets. In our calculus, these are the free names (unbound by input or ν) in P_A. To model other capabilities obtained by corrupting an agent, e.g., database access, we allow for an auxiliary process C'_A to be specified.

Definition 5 (accountability protocol)**.** *Assume a set of parties $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ and $\mathcal{T} \subseteq \mathcal{A}$. An accountability protocol is a ground process of the following form: $\nu\vec{a}; (P_{A_1} \mid C_{A_1} \mid \cdots \mid P_{A_n} \mid C_{A_n})$ where C_{A_i} is of form* event $Corrupt(A_i)$; out($'c', \langle a_1, \ldots, a_m\rangle$); C'_{A_i} *and $\{a_1, \ldots, a_m\} \subseteq \vec{a}$ are the free names in P_{A_i} if $A_i \notin \mathcal{T}$, and 0 otherwise.*

Example 7. *The centralized monitor protocol from Examples 1 and 6 is an accountability process*

$$D \mid (\text{event } Corrupt('D'); \text{out}(sk('D')) \mid M.$$

Processes accessing the store can specify auxiliary processes C_{A_i}, as per default, SAPiC does not permit the adversary to emit events or access the store. With these formal requirements, we can define the set of corrupted parties of a trace as $corrupted(t) = \{A \in \mathcal{A} \mid Corrupt(A) \in t\}$.

The accountability mechanism is defined through the accountability protocol itself and the verdict function. We require the verdict function to be invariant w.r.t. E.

Example 8. *The verdict function for the centralized monitor protocol from Examples 1 and 6, which we sketched in Example 3, can be specified as:*

$$verdict(t) = \begin{cases} \{\{D\}\} & \text{if } t \vDash \omega_1 \\ \varnothing & \text{if } t \vDash \omega_2 \end{cases}$$

for $\omega_1 := \exists a, i, j. Execute(a)@i \wedge LogD(a)@j \wedge a \neq NormalAct$ *and* $\omega_2 := \neg\omega_1$. *As only* $\{\{D\}\}$ *is to be blamed in this example, this verdict function achieves accountability for*

$$\varphi := \forall a', i'.\ Execute(a')@i' \implies a = NormalAct,$$

even w.r.t. the weakest relation r_w. *This can be shown automatically by verifying the verification conditions for* r_w *(Def. 3), i.e.:*

- *exhaustiveness and exclusiveness,*
- *sufficiency of* $\{\{D\}\}$ ($\mathrm{SF}_{\omega_1,\varphi,\{\{D\}\}}$)*: there is a trace* t *s.t.* $t \vDash \exists i. Corrupt(D)@i \wedge \neg(\forall j, a.\ Execute(a)@j \implies a = NormalAct)$*, i.e., a corrupt* D *is able to execute an exceptional action.*
- *verifiability w.r.t.* ω_1 ($\mathrm{V}_{\omega_1,V_1}$)*:* $\omega_1 \implies \neg\varphi$*, which holds a priori.*
- *verifiability w.r.t.* ω_2 ($\mathrm{V}_{\omega_2,V_2}$)*:* $\omega_2 \implies \varphi$*, i.e., the absence of a log for an exceptional action means none was effectuated.*
- *minimality of* $\{\{D\}\}$ (M_{φ,V_1})*:* $\varphi \vee \exists i. Corrupt(D)@i$*. Unless* D *was corrupted, no exceptional action was effectuated.*
- *uniqueness of* $\{\{D\}\}$ ($\mathrm{U}_{\omega_1,V_1}$)*:* $\omega_i \implies \exists i. Corrupt(D)@i$*. An entry in the log blaming* D *can only occur if* D *was actually corrupted.*

VI. Case studies for r_w

In this section and Section VIII, we demonstrate the feasibility of our verification approach on various case studies in different settings. We first concentrate on cases where the weakest counterfactual relation r_w is sufficient, including practical examples like Certificate Transparency and OCSP-Stapling.

We implemented our translation from accountability properties to conditions in SAPiC[5], which provides support for arbitrary relations (leaving the proofs for $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ and $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$ to the user), the relation r_c (as described in Section VII) and the weakest possible relation r_w (Section III). Our fork retains full compatibility with the classic SAPiC semantics, with the extension for liveness properties [6] and operates without any substantial changes to Tamarin. By default, our fork preserves multiset rewrite rules contained in its input, and can thus also serve as a preprocessor for accountability protocols encoded in Tamarin's multiset rewriting calculus.

Our findings are summarized in Table III. For each case study, we give the type (✓ for successful verification, ✗ if we discovered an attack), the number of lemmas generated by our translation, the number of additional helping lemmas[6] and the time needed to verify *all* lemmas (even if an attack was found). Verification was performed on a 2,7 GHz Intel Core i5 with 8GB RAM.

TABLE III
Case studies and results.

protocol	type	# lemmas generated	# helping lemmas	time
Whodunit				
faulty	✗,r_w	16	0	395s
fixed	✓,r_w	8	0	112s
Certificate Transp.				
model from [10]	✓,r_w	31	0	41s
extended model	✓,r_w	21	0	50s
OCSP Stapling				
trusted resp.	✓,r_w	7	3	945s
untrusted resp.	✗,r_w	7	3	12s*
Centralized monitor				
faulty	✗,r_c	17	0	5s
fixed	✓,r_c	17	0	3s
replication	✓,r_c	17	0	7s
Causality				
Desert traveller	✓,r_c	16	0	7s
Early preempt.	✓,r_c	16	0	1s
Late preempt.	✓,r_c	16	0	13s
Accountable alg.				
modified-1	✓,r_c	27	1	5792s
modified-2	✓,r_c	27	1	2047s

(*): time to find counter-examples for $\mathrm{V}_{\omega_i,V_i}$ and $\mathrm{U}_{\omega_i,V_i}$.

```
S := in (a); out(c_SA,a); out(c_SJ,a)
A := in(c_SA,a); out (c_AJ,a)
J := in(c_SJ,a1); in(c_AJ,a2); if a1=a2 then event Equal()
                                        else event Unequal()
ν c_SA; νc_AJ; νc_SJ; (S |A |J |
 !(in(<'corrupt',x>); event Corrupted(x); out(sk(x));
      ( if x='S' then out(c_SA); out (c_SJ))
    |( if x='A' then out(c_SA); out (c_AJ)))))
```

Fig. 2. Whodunit protocol[24, Ex. 8].

Toy example: whodunit: The example in Figure 2 illustrates the difference between verdicts larger than 2 and uncertainty about the correct verdict. Two parties, S and A, coordinate on some value chosen by S. S sends this value to A and to a trusted party J. Then, A is supposed to forward this value to J. We are interested in accountability for J receiving the same value from S and A.

The crux here is that a correct verdict function cannot exist: if J receives different values, there are two minimal explanations for her. Either A altered the value it received or S gave A a different value in the first place. Indeed, if

[5]Currently available in the development branch of Tamarin and to be included in the next release: https://github.com/tamarin-prover/tamarin-prover.

[6]SAPiC, as well as tamarin, are sound and complete, but the underlying verification problem is undecidable [1]. Therefore, analyses in SAPiC/Tamarin sometimes employ *helping lemmas* to help the verification procedure terminate. Just like security properties, they are stated by the user and verified by the tool.

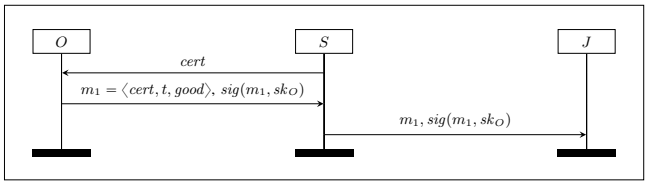

Fig. 3. OCSP Stapling

we formalize this in two verdict functions, one blaming A if the fact $Unequal$ occurs in the trace, the other blaming S, Tamarin finds a counterexample for each. If we change the protocol so that S needs to sign her message, and A needs to forward this signature, then we can prove accountability for the verdict function that blames S in case of inequality.

Certificate Transparency: Certificate Transparency [35] is a protocol that provides accountability to the public key infrastructure. Clients are submitting certificates signed by CAs to logging authorities (LAs), who store this information in a Merkle tree. Auditors validate that these logs have not been tampered with. Based on these trustworthy, distributed logs, clients, e.g., domain owners, may detect misbehavior, e.g., an unauthorised CA, issuing certificates for this domain.

We base our modelling on that of Bruni et.al. [10], which considers a simplified setting with one CA, one LA, and two honest auditors. We first verify accountability of the CA for the property that any certificate in the log that is tested was honestly registered. To this end, we, as well as Bruni et.al., have to assume access to the CA's domain registration data. We then verify accountability of the LA for the property that any log entry that was provided was provided consistently to all auditors. Finally, we compose both security properties and verdict functions, and can thus show that CA and LA can be held accountable at the same time.

While the original modelling prescribed cheating LAs to cheat in a certain way (always provide the correct log entry to auditor u_1 and omit it to auditor u_2), we extended the model to permit deviating LAs to selectively provide log entries. This complicates the formulation of the consistency criterion, but makes the model slightly more realistic. Both models can be verified within a minute.

OCSP Stapling: The Online Certificate Status Protocol (OCSP [34]) provides an interface to query whether a digital certificate is revoked, or not. Upon a request (which may be signed or not), a trusted responder (e.g., a CA or a delegate) gives a signed response containing a timestamp, the certificate in question and its revocation status (*good*, *revoked* or *unknown*). *OCSP Stapling* [33], is an extension to TLS that specifies how a TLS server may attach a recent enough OCSP response to a handshake. This reduces the communication overhead. In addition, it avoids clients exposing their browsing behavior to the OCSP server via their requests.

We model OCSP stapling as an accountability protocol between a trusted OCSP responder O, an untrusted server S and a trusted Judge J. The judge represents a client that receives a stapled response from the server and seeks to determine if its communication partner can be trusted. In addition, a clocking process emits timestamps. Our modelling is quite simplistic, e.g., the TLS Handshake is reduced to a forwarding of the signed timestamp. The main challenge we focussed on was defining the accountability property that is actually achieved. First, note that a server can choose to reveal its secret key at any time. In order to make any meaningful statement about the revocation mechanism, we have to limit ourselves to cases where, whenever a server reveals his secret, it also revokes the corresponding certificate. We thus slightly diverge from the corruption procedure in Definition 5, and require the server to mark his certificate as revoked upon corruption.

We can show accountability for $\varphi =$

$$\neg\exists c, sk, t, i, j, k, l. Judged(t, c)@i \wedge Secret(c, sk)@j \\ \wedge K(sk)@k \wedge Time(t)@l \wedge k < l,$$

i.e., whenever a client received an OCSP response for a certificate c with timestamp t (at which point $Judged(t, c)$ is raised), she can be assured that the corresponding secret sk was not leaked (recall that $K(sk)$ marks the adversary sending a message) at a point in time k prior to the emission of the timestamp t at time l (but possibly later). Timestamps are here modelled as public nonces emitted by the clock process. Prior to outputting a nonce, the event $Time(t)$ is emitted, binding these nonces to positions in the trace.

To explore the limits of OCSP, we declare the OCSP responder to be untrusted. We find an attack on the previous accountability property where O reports a *good* certificate status to J, despite the revocation triggered when corrupting S. The techniques used by Milner et.al. to detect misuse of secrets [32] could potentially be used to mitigate this issue. We leave this research question for future work.

VII. Verifying RL and RS

For protocols in which parties can collude, but can also cause damage on their own, we need to verify accountability with respect to a relation r between what actually happened, and what could have happened with a subset of these parties (see Example 2). The verification conditions in Section IV provide a framework for many counterfactual relations r. In this section, we show how r can be instantiated to the relation r_c discussed in Section II, so that the conditions $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ and $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$, and thus accountability as a whole, can be verified with SAPiC and Tamarin. For other relations, which includes relations that cannot be formally stated such as r_k, $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ and $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$ need to be proven or justified on paper (but the remaining conditions can be verified).

In practice, the control-flow of a process is not necessarily the control-flow of its implementation. To leave some degree of flexibility to the modeller, we allow for the control-flow to be manually annotated and use a unary fact $Control \in \Sigma_{fact}$ to mark control-flow. Per default, there should be exactly one statement event $Control(p : pub)$ on each path from the root to a leaf of the process tree of each process corresponding to

a trusted party. We can then define $r_c(t, t')$ iff for all p and p' s.t. $Control(p) \in t$ and $Control(p') \in t'$, $p =_E p'$.

The main challenge in proving $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ and $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$ is that SAPiC supports only trace properties.[7] Hence, in general, we can argue about all or some t, but not about pairs of t and t'. The solution is to combine t and t' in a single trace, which is their concatenation. If for all occurrences of $Control(p)$ in the first part, and of $Control(p')$ in the second part, p and p' coincide, then the t and t' corresponding to these two parts are in the relation. This technique is known as self-composition [7].

Defining this sequential composition of P with itself is technically challenging and requires altering the translation from SAPiC to multiset rewrite rules — observe that $P; P$ is not a syntactically valid process. Due to space limitations, we refer to Appendix E in the full version [23] and F in the full version [23] for the technical solution and its proof of correctness (it is sound and complete), and will only discuss the idea and limitations.

The idea is that the adversary can start executions with a fresh execution id, which is used to annotate visible events. To separate executions, the adversary can trigger a stopping event. A rewriting of the security property ensures that it is trivially true, unless both executions are properly separated, i.e., every execution is terminated, events are enclosed by start and stop events, and these events themselves define disjoint intervals.

Note that for a process where a trusted party is under replication, it is possible that two different $Control()$-events are emitted in the same execution, and thus the corresponding process is in no equivalence class w.r.t. r_c. This affects only one of our case studies (centralized monitor), however, instead of considering a possibly replicating series of violations, we chose to identify the party causing the *first* violation. Appendix E in the full version [23] discusses other possible solutions to this issue in more detail.

VIII. Case studies for r_c

The most challenging protocols from an accountability perspective are those in which joint misbehaviour is possible. The centralized monitoring mechanism from Example 2 provides such an example, as well as the accountable algorithm's protocol proposed by Kroll. In both cases, an analysis w.r.t. a more restrictive counterfactual relation is strictly necessary. We opt for the relation r_c, relating runs having the same control-flow. To this end, we use the elaborate verification condition for arbitrary r (Def. 4) and automate the analysis of $\mathrm{RL}_{R,\omega_i,\omega_j,V_i,V_j}$ and $\mathrm{RS}_{R,V_i=\{\mathrm{S}\}}$ by considering the sequential self-composition of the protocol as described in the previous section. Owing to our accountability definition's origins in causation, we will discuss examples of 'preemption' from the causation literature, which are considered difficult to handle, but can be tamed by considering the control-flow of execution.

[7] Tamarin supports diff-equivalence [8]. This variant of observational equivalence considers two processes unequal if they move into different branches, hence it is not suitable for our case.

```
D:=in(a);
   if isNormal(a) then
      out(⟨m1 := ⟨'Do', a⟩,sign(m1,sk('D'))⟩)
   else if isSpecial (a) then
      out(⟨m2 := ⟨'Permit', a⟩,sign(m2,sk('D'))⟩))
C:=in(⟨m2 := ⟨'Permit', a⟩,m2s⟩);
     if verify (m2s, m2, pk(sk('D')))= true() then
        if isSpecial (a) then
          out(⟨m3 := ⟨m2, m2s⟩,sign(m3,sk('C'))⟩)
M:=
 (in(⟨m1 := ⟨'Do', a⟩,m1s⟩);
  if verify (m1s,m1,pk(sk('D')))=true() then
  event Control('0','1');event LogD(a);event Execute(a))
+(in(⟨m3 := ⟨m2, m2s⟩,m3s⟩); // for m2 := ⟨'Permit', a⟩
  if verify (m3s, m3, pk(sk('C'))) = true() then
  if verify (m2s, m2, pk(sk('D'))) = true() then
  event Control('0','2');event LogDC(a);event Execute(a))

!(D |C) |M
| // give access to public keys
   (out(pk(sk('D')));out(pk(sk('C')));out(pk(sk('M'))))
|!(in('c',⟨'corrupt',x⟩);
    event Corrupted(x); out('c', sk(x))))
```

Fig. 4. Centralized monitor.

Note that we omit code listings for most examples, however, they come with the implementation.

Centralized monitor: Example 2 considered a protocol based on a central trust monitor M. Albeit modelled in a very abstract form (the actual protocols are likely to be tailored for their use case), this kind of mechanism occurs in various forms in plenty of real-world scenarios. We want to demonstrate that, in principle, we can handle such scenarios.

A party D can effectuate actions, some of which are usual (e.g., a doctor requesting his patient's file), some of which are not (e.g., requesting the file of another doctor's patient). Rather than blocking access for exceptional action, these are logged by M, (e.g., if another doctor's patient has a heart attack and needs treatment right away), which is an accountability problem w.r.t. the property that no exceptional action happened. A supervisor C (e.g., chief of medicine) is needed to effectuate a third class of actions, special actions, for which D needs to get C's authorization. The processes of parties D, C and M are running in parallel with a process that outputs their public keys and their private keys on request (see Fig. 4).

We use function symbols $NormalAct/0$, $SpecialAct/0$, $ExcAct/0$ to denote these kinds of actions, and $sig/2$, $verify/2$ to model digital signatures.

D receives an action a from the adversary (for generality) and either signs it and sends it to M directly (if a is 'normal'), or signs a permission request, which C has to sign first (if a is 'special'). C only signs requests for special actions. M non-deterministically guesses what kind of message arrives, and verifies that the signatures are correct. If this is the case, the action is executed.

We investigate accountability for the property that only

'special' or 'normal' actions are executed:

$$\forall a, i.\mathit{Execute}(a)@i \\ \implies a = \mathit{SpecialAct}() \vee a = \mathit{NormalAct}()$$

Within 3 seconds, our toolchain shows that the verdict function that maps the occurrence of LogD to $\{\{D\}\}$ and the occurrence of LogDC to $\{\{D, C\}\}$ provides accountability for this property.

The first protocol design was (without intention) erroneous. We find two attacks within 5 seconds (for falsification and verification of all lemmas).

For simplicity, M is not covered under replication. Putting M under replication, it is possible to have two different $\mathit{Control}$-events in the same run, which entails that no second trace can relate via r_c. We can, however, prove a modified property φ' which is true only if there are no violations to φ, or *at least* two. Intuitively, this means that we can point out which party caused the *first* violation by considering, for every violating trace, the prefix which contains only one violation. This is possible for all safety-properties, as they are, by definition, prefix-closed [28]. In addition, we modify M to only emit $\mathit{Control}$-events when acting upon an action that is neither normal nor special, i.e., we consider only control-flow for actions which produce violations.

Examples from causation literature: As our accountability definition is rooted in causation, we chose to model three examples from the causation literature, two from Lewis' seminal work on counterfactual reasoning [29], and one formulated by Hitchcock [18, p. 526]. They all encode problems of *preemption*, where a process that would cause an event (e.g., a violation) is interrupted by a second processes. All examples are verified in a couple of seconds. We refer to Appendix G in the full version [23] for details.

Kroll's Accountable Algorithms protocol: The most interesting case study is the accountable algorithms protocol of Kroll [21, Chapter 5]. It lets an authority A, e.g., a tax authority, perform computations for a number of subjects S_1 to S_n, e.g., the validation of tax returns. Any subject can verify, using a public log, that the value it receives has been computed correctly w.r.t. the input it provides and some secret information that is the same for all parties. We substantially extend this protocol to also provide accountability for the subjects: if a subject decides to falsely claim that the authority cheated, we can validate their claim. This is a very instructive scenario, as now any party emitting logs or claims can just lie. It also demonstrates that a trusted *third* party is not needed to provide accountability. While we define a judgment process J, which is technically a trusted party (it is always honest), this party is (a) not involved in the protocol and (b) can be computed by anyone, e.g., a journalist, an oversight body, or the general public.

The goal is to compute a function f on inputs x and y; f representing an arbitrary algorithm, x the subject's input, and y some secret policy that f takes into account (e.g., an income threshold to decide which tax returns need extra scrutiny). The policy y is the same for all subjects. First, the authority commits on y (see Figure 5). We model commitments using the symbols $\mathit{commit}/2$, $\mathit{open}/2$, $\mathit{verCom}/2$ and equations $\mathit{open}(\mathit{commit}(m, r), r) = m$ and $\mathit{verCom}(\mathit{commit}(m, r), r) = \mathit{true}()$. Next, the subject sends its input x along with randomness for its desired commitment r'_x and a signature for the commitment $C'_x := \mathit{commit}(x, r'_x)$. Now the authority computes $z = f(x, y)$ and returns z, as well as the randomness which was used to compute two commitments on its own, C_x on x and C_z on z. The signed commitment C'_x and a zero-knowledge proof (ZKP) are stored in a public append-only log, e.g., the blockchain. The log is modelled via the store and a global assumption that entries cannot be overwritten. The ZKP contains the three commitments C_x, C_y and C_z and shows that they are commitments to x, y and z such that $f(x, y) = z$. Using this ZKP, one can check that C_y is the value initially committed to, and that the input x and the output z are consistent. Now A proceeds with the next subject, and appends a Final message to the log to indicate when it is done. S can decide to file a claim, consisting of x, z, r'_x and the signed message it received from A in the second step.

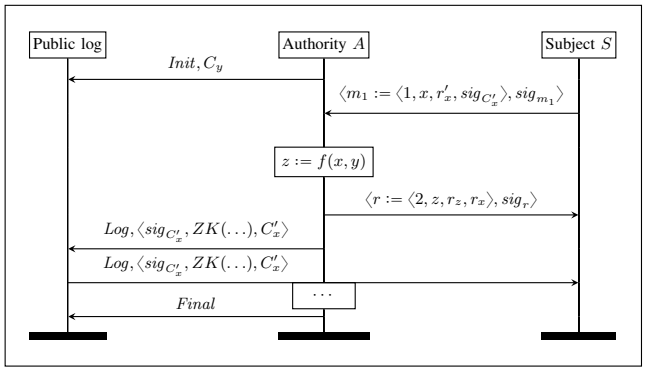

Fig. 5. Honest protocol run for accountable algorithm protocol.

The original protocol was implemented with hash functions and ZK-SNARKs. Kroll's preliminary analysis [21, Section 5.2.2] was informal and only considered holding the authority accountable. Later analysis discovered that any subject can falsely accuse the authority A of misbehavior [24]. Obviously, such a claim would subvert the trust into the system and hence render the accountability mechanism useless in practice. Consequently, we extended the protocol with a message authentication mechanism based on digital signatures so that A and S_i prove their identity and protect themselves from false accusations, as well as C'_x, which is a second commitment on x generated by S instead of A.

This commitment (along with S'es signature) serves as a witness that the input which A claims to have received is indeed the one S has sent, without revealing x to the public.

On the other hand, we simplified Kroll's protocol by removing randomness generation for f, which was implemented using verifiable random functions. We outline in Section X how this feature calls for a definition of accountability in the

```
S:= let res    = ⟨'2', z, rz, rx⟩
       claim   = ⟨'3', x, z, res, sig_res, rxp⟩
       Cxp     = commit(x, rxp)
       m1      = ⟨'1', x, rxp, sign(Cxp, sk('S'))⟩
       sig_cxp = fst(log)
       zkp     = fst(snd(log))
       CxpL    = snd(snd(log))
       Cy      = fst(snd(Pub(zkp)))
  in
  lookup ⟨'A', ' Init '⟩ as CyL in
  ν x; ν rxp; out (⟨m1, sign(m1,sk('S'))⟩);
  lookup ⟨'A','Log','S'⟩ as log in
   if and4( verZK(zkp), eq(sig_cxp,sign(Cxp, sk('S'))),
            eq(CxpL,Cxp), eq(Cy, CyL)) then
                in (⟨res, sig_res⟩);
                 if verify(sig_res, res, pk(sk('A')))=true()
                     then
                  out (⟨claim, sign(claim, sk('S'))⟩)

A:= let m1 = ⟨'1', x, rxp, sig_cxp⟩
       z       = f(x, y)
       res     = ⟨'2', z, rz, rx⟩
       Cx        = commit(x, rx)
       Cy        = commit(y, ry)
       Cz        = commit(z, rz)
       Cxp       = commit(x, rxp)
       zkp       = ZK(⟨Cx,Cy,Cz⟩,x,y,z,rx,ry,rz)
       sig_res   = sign(res, sk('A'))
  in
  ν y; ν ry;
  insert ⟨'A', ' Init '⟩, Cy;
  in (⟨m1, sig_m1⟩); ν rx; ν rz;
   if and3( verify(sig_m1,m1,pk(sk('S'))),
       eq(x, open(Cxp,rxp)),
        verify(sig_cxp,Cxp,pk(sk('S')))) then
       out (⟨res, sig_res⟩);  // send result to S
       insert ⟨'A','Log','S'⟩, ⟨sig_cxp, zkp, Cxp⟩;
       insert ⟨'A','Final'⟩, true()

!(A |S) |J |
  out(pk(sk('A'))); out(pk(sk('S')))
|
 !(in('c',⟨'corrupt',x⟩);event Corrupted(x);out('c',sk(x));
   !(( if x='A' then in(y); insert ⟨'A',' Init '⟩,y)
    |( if x='A' then in(y); insert ⟨'A','Log','S'⟩, y)
    |( if x='A' then in(y); insert ⟨'A','Final'⟩, y))
    |( if x='A' then lookup ⟨'A','Log','S'⟩ as y in out(y))
    |( if x='S' then lookup ⟨'A','Log','S'⟩ as y in out(y))
    ))
```

Fig. 6. Accountable algorithms protocol.

```
J:= let res    = ⟨'2', z, rz, rx⟩
       claim   = ⟨'3', x, z, res, sig_res, rxp⟩
       sig_cxp = fst(log)  // log is ⟨sig_cxp, zkp, Cxp⟩;
       zkp     = fst(snd(log))
       CxpL    = snd(snd(log))
       Cx      = fst(Pub(zkp))
       Cy      = fst(snd(Pub(zkp)))
       Cz      = snd(snd(Pub(zkp)))
in
in (⟨claim, sig_claim⟩);
if verify(sig_claim, claim, pk(sk('S'))) = true() then
  lookup ⟨'A','Final'⟩ as y in
  event Final();
  lookup ⟨'A',' Init '⟩ as CyL in
  event Comm(CyL);
  lookup ⟨'A','Log','S'⟩ as log in
  // first check validity of the log by itself,
  if and3(verZK(zkp), eq(CyL, Cy), //produced by A
     verify(sig_cxp,CxpL,pk(sk('S')))) // verified by A
  then
       if and3( verify(sig_res,res,pk(sk('A'))), // honest S
            verifies this
              verCom(CxpL,rxp), // produced by S
              eq(x, open(CxpL,rxp))) // both signed by S
       then // We now believe S is honest and its claim valid
          if and4(verCom(Cx,rx), verCom(Cz,rz),
                  eq(x, open(Cx, rx)),eq(z, open(Cz, rz)))
          then
            event Control('0','1');event HonestS();
            event HonestA();event Verified(claim)
          else
             if oracle(x,Cy,z)=true() then // see below.
               event Control('0','2');event HonestS();
               event HonestA(); event Verified(claim)
             else
               event Control('0','3');event HonestS();
               event DisHonestA(); event Verified(claim)
       else  // A's log is ok, but S is definitely cheating
          if oracle(x,Cy,z)=true() then
            event Control('0','4');event HonestS();
            event HonestA(); event Verified(claim))
          else
            event Control('0','5');event DisHonestS();
            event HonestA();event Verified(claim)
  else (// A is dishonest and produced bad log
     if oracle(x,CyL,z)=true() then
       event Control('0','6');event HonestS();
       event HonestA();event Verified(claim)
     else
       event Control('0','7');event DisHonestA();
       event DisHonestS();
       // S checks log before submitting claim→ S dishonest
       event Verified(claim))
```

Fig. 7. Accountable algorithms protocol: judging procedure.

computational setting.

We model this protocol as an accountability process with three subprocesses A, S and a judging procedure J (see Figure 6 and 7). This judging procedure essentially determines the verdict by inspecting the log and a claim output by S. For brevity, we introduce a symbol $eq/2$ with equation $eq(x, x) = true$ and predicates $and3$ and $and4$ which evaluate to true if all three or four terms equal the constant $true$. We also use syntactic sugar let $v = t$ in P to denote the literal substitution of v by t in P.

The verdict function is a trivial conversion of the events emitted by the judging procedure J, which inspects the log and evaluates the claim for consistency (see Figure 7). We decided for this approach in order to ensure that the judgement can be made by the public, and to provide an algorithmic description on how to do so, since a verdict function can easily be written to rely on information that is not available or not

computable. J is technically a trusted party in the sense that it is always honest. However, J is not involved in the protocol and the fact can be computed after the fact by anyone who receives the claim, e.g., a newspaper, an oversight body, or the general public. A dishonest J merely means that the verdict function is computed incorrectly, in which case we cannot, and should not, make any guarantees. The verdict function maps traces in which $DishonestA$ and $DishonestS$ appear to be $\{\{A, S\}\}$, and traces with $DishonestA$ and $HonestS$ to be $\{\{A\}\}$. It provides the protocol with accountability for the following property φ:

$$\forall t, x, z, r, sig_r, r_{x'}. \mathit{Verified}(\langle t, x, z, r, sig_r, r_{x'}\rangle)@k \\ \implies \exists j, i, y, r_y. \mathit{Final}()@i \wedge \mathit{Comm}(\mathit{commit}(y, r_y))@j \\ \wedge z = f(x, y) \wedge i < j < k.$$

This property guarantees that any claim considered by the judging procedure is correctly computed w.r.t. f and the initial commitment to y. Note that a violation requires a $\mathit{Verified}$-event, which J only emits if, and only if, the first conditional and the two subsequent lookups are successful. The conditional formulates the requirement of a claim to be signed by S, the two lookups require A to indicate the protocol is finished and to have produced an Init entry, in order to ensure that the claim is only filed after the protocol has finished.

The judging procedure needs to rely on an external oracle to determine whether a violation actually took place. This is a restriction, as it requires any party making a judgement to have access to such an oracle. We need this for the following reason: accountability implies verifiability, hence, even if the logs have been tampered with, and they do not contain usable information, the verdict function needs to nevertheless output the empty verdict if $z = f(x, y)$. For example, if C_y in the log does not match x' and z in claim, i.e., $z \neq f(x', \mathit{open}(C_y, r_y))$, but the logged C_x is a commitment to a different x, it could well be the case that $z = f(x, y)$. Figuratively speaking, the adversary tries to trick us into thinking something went wrong. We represent the oracle as a function symbol $\mathit{oracle}/3$ with equation $\mathit{oracle}(x, \mathit{commit}(y, ry), f(x, y)) = \mathit{true}$, and restrain ourselves to only use this function in the judging procedure.

The judging procedure is instructive in the constraints it puts on the parties. Broadly speaking, they have to assist J in holding the opposite party accountable by validating incoming messages. E.g., if A manipulates the log by providing an invalid ZKP, S terminates, and φ is trivially true. This ensures that J can count on the validity of the log unless S is dishonest. On the other hand, A can now stall the process. From a protocol design perspective, this raises the following challenges: *1.* Are there guiding principles for the judging procedure? (Our design involved a lot or trial and error, which is only a viable strategy due to the tool support we have.) *2.* Is it possible to achieve accountability for timeliness, i.e., the property that the protocol eventually finishes [6]? *3.* Can we do without oracle access in the judging procedure?

The verification of the modified variant of Kroll's protocol takes about two hours. An alternative way of fixing this model requires a modified zero-knowledge proof but is structurally closer to the original proposal. In order to avoid maintaining both C_x (chosen by A) and $C_{x'}$ (chosen by S), we allow S to chose C_x, i.e., the commitment to x in the public part of the ZKP, similar to how the commitment to y has to match the commitment sent out at the beginning of the protocol. We therefore modelled and verified both fixes, but present only one here. This second variant can be verified in half an hour. In both cases, the analysis is fully automatic, but a helping lemma (which itself is verified automatically) was added for speedup. It states that lookups for the same key in the log always give the same result.

IX. Related work

The focus of this work is on accountability in the security setting; here we understand accountability as the ability to identify malicious parties. Early work on this subject provides or uses a notion of accountability in either an informal way, or tailored to the protocol and its security [3, 21, 4, 5]. The difficulty is defining what constitutes 'malicious behavior' and what this means for completeness, i.e., the ability of a protocol to hold *all* malicious parties accountable. Jagadeesan et al. provided the first generalized notion of accountability, considering parties malicious if they deviate from the protocol. But in their model, 'the only auditor capable of providing [completeness] is one which blames all principals who are capable of dishonesty, regardless of whether they acted dishonestly or not' [19]. Algorithms in distributed systems, e.g., PeerReview [17], use this notion to detect faults in the Byzantine setting, but need a complete local view of all participating agents. Our corruption model is also Byzantine, however, we work in the security setting and thus we do not assume that a complete view of every component or the whole communication is available.

For the security setting, Küsters et al. recognize that completeness according to this definition of maliciousness cannot be fulfilled (while remaining sound), because it 'includes misbehavior that is impossible to be observed by any other party [or is harmless]'. They propose to capture completeness via policies. Künnemann *et al.* [24] argue that these policies are not expressive enough. In case of joint misbehavior, the policy is either incomplete, unfair, or it encodes the accountability mechanism itself [24]. Other approaches focus on protocol actions as causes for security violations [16, 12, 15]. However, not all protocol actions that are related to an attack are necessarily malicious. Consider, e.g., an attack that relies on the public key retrieved from a key server — the sending of the public key is causally related to the violation, but harmless in itself. While causally related protocol actions can be a filter and a useful forensic tool, they refer us back to the original question: What constitutes malicious misbehavior?

Künnemann *et al.* [24] answer these questions by considering *the fact* that a party deviated as a potential cause [24]. The main difference between their framework and the present is

that they define a calculus which allows for individual parties to deviate. In particular, deviating parties can communicate with each other and make their future behavior dependent on such signals. This allows to encode 'artificial' causal dependencies in their behavior, e.g., a party A, instead of mounting an attack, waits for an arbitrary message from a second party B before doing so. If one only observes A's attack, the involvement of B is as plausible as the idea of A acting on its own. This kind of 'provocation' can occur whenever B can secretly communicate with A. As out-of-band channels cannot be excluded in practice, this shows that accountability is impossible. In the single-adversary setting, provocation cannot be encoded: all deviating parties are represented by the adversary; neither B sending the provocation message to A, nor A conditioning the attack on the arrival of this message can be expressed. As the case studies show, the provocation problem vanishes. This implies that the single-adversary setting comes at a loss of generality — although it is a commonly assumed worst-case assumption in security. They propose 'knowledge-optimal' attacks, where only secret information is exchanged, which we *conjecture* to be equivalent to the single-adversary setting following [24, Lemma 3].

Independently, Bruni, Giustolisi and Schürmann, propose a definition of accountability that is based on parties choosing to deviate [10]. In contrast to our verdict function, they consider per-party tests, e.g., 'Is party A to be held accountable in this run'. Any set of these tests can be interpreted as a verdict function that outputs the set of singleton sets for which the test gave true. In fact, this is how we constructed the verdict function in the Certificate Transparency case study from their tests. The downside to this approach is that joint accountability is not expressible, excluding, e.g., the case studies in Section VIII. Nevertheless, it is instructive to compare their definition to our verification conditions for r_w (other relations are not covered). The definition has four criteria, three of which have logically equivalent formulae in our set of verification conditions. Their fourth criterion, however, is weaker (than $\mathrm{V}_{\omega_i,V_i}$), and our criterion $\mathrm{SF}_{\omega_i,\varphi,\mathrm{S}}$ is missing in their definition. We argue that both conditions are strictly necessary: without $\mathrm{SF}_{\omega_i,\varphi,\mathrm{S}}$, a protocol that does not even permit a violation (φ is always true), could fulfill all their criteria and still (unfairly!) blame a party. Furthermore, their counterpart to $\mathrm{V}_{\omega_i,V_i}$ allows violations to remain undetected under certain circumstances. We elaborate this in Appendix H in the full version [23]. This confirms our top-down approach: by building on accountability as a problem of causation, we were able to ground our verification conditions in Def. 1. We are able to specify exactly what our verdict function is computing and can be sure to not forget conditions.

X. Conclusion and future work

We demonstrated the practicality of verifying accountability in security protocols with a high degree of automation, and thus call for the analysis of existing and the invention of new protocols that provide for accountability, instead of blindly trusting third parties, e.g., voting protocols.

The present definitions apply to a wide-range of protocol calculi. We implemented support for the SAPiC calculus, which allows for a precise definition of control-flow. However, the definition we provide cannot express computational or statistical indistinguishability. For this reason, we had to simplify Kroll's accountable algorithms protocol, and ignore its ability to randomize the computation using a verifiable random function, and thus implement, e.g., accountable lotteries. A computational variant of our definition would be desirable, but it poses some technical challenges. In particular, counterfactual adversaries shall not depend on the randomness actually used, which prohibits a straight-forward translation and thus renders the formulation of a generalized accountability game an interesting question for future research.

Acknowledgements: We thank Deepak Garg and Rati Devidze for discussion and advice in the early stages of this work, and Hizbullah Abdul Aziz Jabbar for help on the accountable algorithms protocol. This work has been partially funded by the German Research Foundation (DFG) via the collaborative research center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223), and via the ERC Synergy Grant IMPACT (Grant agreement ID: 610150). The second author holds the Ministry of National Education Scholarship of the Turkish Republic.

References

[1] Martín Abadi and Véronique Cortier. "Deciding knowledge in security protocols under equational theories". In: *Theoretical Computer Science* 387.1-2 (2006).

[2] Martín Abadi and Cédric Fournet. "Mobile Values, New Names, and Secure Communication". In: *POPL*. 2001.

[3] N. Asokan, Victor Shoup, and Michael Waidner. "Asynchronous protocols for optimistic fair exchange". In: *Security and Privacy*. 1998.

[4] Michael Backes, Jan Camenisch, and Dieter Sommer. "Anonymous yet accountable access control". In: *Workshop on Privacy in the Electronic Society*. 2005.

[5] Michael Backes, Dario Fiore, and Esfandiar Mohammadi. "Privacy-Preserving Accountable Computation". In: *ESORICS*. 2013.

[6] Michael Backes et al. "A Novel Approach for Reasoning about Liveness in Cryptographic Protocols and its Application to Fair Exchange". In: *EuroS&P*. 2017.

[7] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 2004.

[8] David A. Basin, Jannik Dreier, and Ralf Sasse. "Automated Symbolic Proofs of Observational Equivalence". In: *22nd Conference on Computer and Communications Security (CCS'15)*. 2015.

[9] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *Computer Security Foundations Workshop*. 2001.

[10] Alessandro Bruni, Rosario Giustolisi, and Carsten Schürmann. "Automated Analysis of Accountability". In: *ISC 2017*. 2017.
[11] Cas J.F. Cremers. "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols". In: *20th Conference on Computer Aided Verification (CAV'08)*. 2008.
[12] Anupam Datta et al. "Program actions as actual causes: A building block for accountability". In: *CSF*. 2015.
[13] Joshua Dressler. *Understanding criminal law*. Matthew Bender, 1995.
[14] Santiago Escobar, Catherine Meadows, and José Meseguer. "Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties". In: *Foundations of Security Analysis and Design*. 2009.
[15] Joan Feigenbaum, Aaron D. Jaggard, and Rebecca N. Wright. "Towards a Formal Model of Accountability". In: *New Security Paradigms Workshop*. 2011.
[16] Gregor Gößler and Daniel Le Métayer. "A general framework for blaming in component-based systems". In: *Sci. Comput. Program.* 113 (2015).
[17] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. "PeerReview: Practical Accountability for Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007).
[18] Christopher Hitchcock. "Prevention, Preemption, and the Principle of Sufficient Reason". In: *Philosophical Review* 116.4 (2007).
[19] Radha Jagadeesan et al. "Towards a theory of accountability and audit". In: *ESORICS*. 2009, pp. 152–167.
[20] Steve Kremer and Robert Künnemann. "Automated analysis of security protocols with global state". In: *Journal of Computer Security* 24.5 (2016).
[21] Joshua A. Kroll. "Accountable Algorithms". PhD thesis. Princeton University, 2015.
[22] Robert Künnemann. "Sufficient and necessary causation are dual". In: *CoRR* abs/1710.09102 (2017).
[23] Robert Künnemann, Ilkan Esiyok, and Michael Backes. "Automated Verification of Accountability in Security Protocols". In: *CoRR* abs/1805.10891 (2018).
[24] Robert Künnemann, Deepak Garg, and Michael Backes. *Accountability in Security Protocols*. Cryptology ePrint Archive, Report 2018/127. 2018.
[25] Robert Künnemann, Deepak Garg, and Michael Backes. "Causality & Control flow". In: *4th Workshop on Formal Reasoning about Causation, Responsibility, & Explanations in Science & Technology*. to appear. 2019.
[26] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. "From Probabilistic Counterexamples via Causality to Fault Trees". In: *Computer Safety, Reliability, and Security*. 2011.
[27] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. "Accountability: Definition and Relationship to Verifiability". In: *CCS*. 2010.
[28] Leslie Lamport. "Proving the correctness of multiprocess programs". In: *Transactions on Software Engineering* 2 (1977).
[29] David Lewis. "Causation". In: *Journal of Philosophy* 70.17 (1973).
[30] Gavin Lowe. "A hierarchy of authentication specifications". In: *Computer Security Foundations Workshop*. 1997.
[31] J. L. Mackie. *The Cement of the Universe: A Study of Causation*. Clarendon Press, 1980.
[32] K. Milner et al. "Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications". In: *CSF*. 2017.
[33] D. Eastlake 3rd. *Transport Layer Security (TLS) Extensions: Extension Definitions*. RFC 6066 (Proposed Standard). 2011.
[34] S. Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960 (Proposed Standard). 2013.
[35] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962 (Experimental). 2013.
[36] Benedikt Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *CSF*. 2012.

APPENDIX

A. Operational semantics

Frames and deduction: Before giving the formal semantics of SAPiC, we introduce the notions of frame and deduction. A *frame* consists of a set of fresh names $\tilde{n}$ and a substitution σ, and is written $\nu\tilde{n}.\sigma$. Intuitively, a frame represents the sequence of messages that have been observed by an adversary during a protocol execution and secrets $\tilde{n}$ generated by the protocol, a priori unknown to the adversary. Deduction models the capacity of the adversary to compute new messages from the observed ones.

Definition 6 (Deduction)**.** *We define the deduction relation $\nu\tilde{n}.\sigma \vdash t$ as the smallest relation between frames and terms defined by the deduction rules in Figure 8.*

Operational semantics: We can now define the operational semantics of our calculus. The semantics is defined by a labelled transition relation between process configurations. A *process configuration* is a 5-tuple $(\mathcal{X}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$ where

- $\mathcal{X} \subseteq FN$ is the set of fresh names generated by the processes;
- $\mathcal{S} : \mathcal{M}_\Sigma \to \mathcal{M}_\Sigma$ is a partial function modeling the store;
- $\mathcal{P}$ is a multiset of ground processes representing the processes executed in parallel;
- σ is a ground substitution modeling the messages output to the environment;
- $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently active locks

The transition relation is defined by the rules in Figure 9. Transitions are labelled by sets of ground facts. For readability, we omit empty sets and brackets around singletons, i.e., we

$$\frac{a \in FN \cup PN \quad a \notin \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\ \textsc{Dname} \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\ \textsc{DEq}$$

$$\frac{x \in dom(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\ \textsc{DFrame} \qquad \frac{\nu\tilde{n}.\sigma \vdash t_1 \cdots \nu\tilde{n}.\sigma \vdash t_n \quad f \in \Sigma^k \backslash \Sigma^k_{priv}}{\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_n)}\ \textsc{DAppl}$$

Fig. 8. Deduction rules.

Standard operations:

$$\begin{aligned}
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{0\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P|Q\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P, Q\}, \sigma, \mathcal{L}) \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{!P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{!P, P\}, \sigma, \mathcal{L}) \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\nu a; P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X} \cup \{a'\}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\{a'/a\}\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } a' \text{ is fresh} \\
(\mathcal{X}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) &\xrightarrow{K(M)} (\mathcal{X}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \quad \text{if } \nu\mathcal{X}.\sigma \vdash M \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{out}(M, N); P\}, \sigma, \mathcal{L}) &\xrightarrow{K(M)} (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{^N/_x\}, \mathcal{L}) \\
&\qquad \text{if } x \text{ is fresh and } \nu\mathcal{X}.\sigma \vdash M \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{in}(M, N); P\}, \sigma, \mathcal{L}) &\xrightarrow{K(\langle M, N\tau \rangle)} (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } \nu\mathcal{X}.\sigma \vdash M, \nu\mathcal{X}.\sigma \vdash N\tau \text{ and } \tau \text{ is grounding for } N \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{out}(M, N); P, \mathsf{in}(M', N'); Q\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{P, Q\tau\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } M =_E M' \text{ and } N =_E N'\tau \text{ and } \tau \text{ grounding for } N' \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{\text{if } pr(M_1, \ldots, M_n) \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } \phi_{pr}\{^{M_1}/_{x_1}, \ldots, ^{M_n}/_{x_n}\} \text{ is satisfied} \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{\text{if } pr(M_1, \ldots, M_n) \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{Q\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } \phi_{pr}\{^{M_1}/_{x_1}, \ldots, ^{M_n}/_{x_n}\} \text{ is not satisfied} \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{\text{event}(F);\ P\}, \sigma, \mathcal{L}) &\xrightarrow{F} (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L})
\end{aligned}$$

Operations on global state:

$$\begin{aligned}
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{insert } M, N;\ P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}[M \mapsto N], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}) \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{delete } M;\ P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}[M \mapsto \bot], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}) \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\ \}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\{V/x\}\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } \mathcal{S}(N) =_E V \text{ is defined and } N =_E M \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\ \}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L}) \\
&\qquad \text{if } \mathcal{S}(N) \text{ is undefined for all } N =_E M \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{lock } M;\ P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \cup \{M\}) \quad \text{if } M \notin_E \mathcal{L} \\
(\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{unlock } M;\ P\}, \sigma, \mathcal{L}) &\longrightarrow (\mathcal{X}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \backslash \{M' \mid M' =_E M\})
\end{aligned}$$

Fig. 9. Operational semantics.

write $\rightarrow$ for $\xrightarrow{\varnothing}$ and $\xrightarrow{f}$ for $\xrightarrow{\{f\}}$. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$ (the transitions that are labelled by the empty sets) and write $\stackrel{f}{\Rightarrow}$ for $\rightarrow^* \xrightarrow{f} \rightarrow^*$. We can now define the set of traces, i.e., possible executions that a process admits.

Definition 7 (Traces of P)**.** *Given a ground process P, we define the* traces of P *as*

$$traces(P) = \Big\{ (F_1, \ldots, F_n) \mid c_0 \stackrel{F_1}{\Longrightarrow}_* \ldots \stackrel{F_n}{\Longrightarrow}_* c_n \Big\},\ \textit{where}$$

$c_0 = (\varnothing, \varnothing, \{P\}, \varnothing, \varnothing, \varnothing)$*, A* progressing trace *additionally fulfils the condition that all processes in $\mathcal{P}_n$ are blocking [6, Def. 2].*

If we are interested in liveness properties, we will only consider the set of *progressing* traces, i.e., traces that end with a *final* state. Intuitively, a state is final if all messages on resilient channels have been delivered and the process is *blocking* [6, Def. 2, 3].

B. Security properties

In the Tamarin tool [36], security properties are described in an expressive two-sorted first-order logic. The sort $temp$ is used for time points, $\mathcal{V}_{temp}$ are the temporal variables.

Definition 8 (Trace formulae)**.** *A trace atom is either false $\bot$, a term equality $t_1 \approx t_2$, a timepoint ordering $i \lessdot j$, a*

timepoint equality $i \doteq j$, or an action $F@i$ for a fact $F \in \mathcal{F}$ and a timepoint i. A trace formula is a first-order formula over trace atoms.

(If clear from context, we use $t_1 = t_2$ instead of $t_1 \approx t_2$, $i < j$ instead of $i \lessdot j$, and $i = j$ instead of$i \doteq j$.)

To define the semantics, let each sort s have a domain $dom(s)$. $dom(temp) = \mathcal{Q}$, $dom(msg) = \mathcal{M}$, $dom(fresh) = FN$, and $dom(pub) = PN$. A function $\theta : \mathcal{V} \to \mathcal{M} \cup \mathcal{Q}$ is a valuation if it respects sorts, i.e., $\theta(\mathcal{V}_s) \subset dom(s)$ for all sorts s. If t is a term, $t\theta$ is the application of the homomorphic extension of θ to t.

Definition 9 (Satisfaction relation)**.** *The satisfaction relation $(tr, \theta) \models \varphi$ between a trace tr, a valuation θ, and a trace formula φ is defined as follows:*

$$
\begin{array}{lll}
(tr,\theta) \models \bot & & \textit{never} \\
(tr,\theta) \models F@i & \iff & \theta(i) \in idx(tr) \wedge F\theta \in_E tr_{\theta(i)} \\
(tr,\theta) \models i \lessdot j & \iff & \theta(i) < \theta(j) \\
(tr,\theta) \models i \doteq j & \iff & \theta(i) = \theta(j) \\
(tr,\theta) \models t_1 \approx t_2 & \iff & t_1\theta =_E t_2\theta \\
(tr,\theta) \models \neg\varphi & \iff & \textit{not } (tr,\theta) \models \varphi \\
(tr,\theta) \models \varphi_1 \wedge \varphi_2 & \iff & (tr,\theta) \models \varphi_1 \textit{ and } (tr,\theta) \models \varphi_2 \\
(tr,\theta) \models \exists x : s.\varphi & \iff & \textit{there is } u \in dom(s) \\
 & & \textit{such that } (tr, \theta[x \mapsto u]) \models \varphi.
\end{array}
$$

For readability, we define $t_1 \gtrdot t_2$ as $\neg(t_1 \lessdot t_2 \vee t_1 \doteq t_2)$ and $(\lessdot, \doteq, \gtrdot)$ as expected. We also use classical notational shortcuts such as $t_1 \lessdot t_2 \lessdot t_3$ for $t_1 \lessdot t_2 \wedge t_2 \lessdot t_3$ and $\forall i \leqslant j.\ \varphi$ for $\forall i.\ i \leqslant j \to \varphi$. When φ is a ground formula we sometimes simply write $tr \models \varphi$ as the satisfaction of φ is independent of the valuation.

Definition 10 (Validity, satisfiability)**.** *Let $Tr \subseteq (\mathcal{P}(\mathcal{G}))^*$ be a set of traces. A trace formula φ is said to be* valid *for Tr (written $Tr \models^\forall \varphi$) if for any trace $tr \in Tr$ and any valuation θ we have that $(tr, \theta) \models \varphi$.*

A trace formula φ is said to be satisfiable *for Tr, written $Tr \models^\exists \varphi$, if there exist a trace $tr \in Tr$ and a valuation θ such that $(tr, \theta) \models \varphi$.*

Note that $Tr \models^\forall \varphi$ iff $Tr \not\models^\exists \neg\varphi$. Given a multiset rewriting system R we say that φ is valid, written $R \models^\forall \varphi$, if $traces^{msr}(R) \models^\forall \varphi$. We say that φ is satisfied in R, written $R \models^\exists \varphi$, if $traces^{msr}(R) \models^\exists \varphi$. Similarly, given a ground process P we say that φ is valid, written $P \models^\forall \varphi$, if $traces(P) \models^\forall \varphi$, and that φ is satisfied in P, written $P \models^\exists \varphi$, if $traces(P) \models^\exists \varphi$.