Program Actions as Actual Causes: A Building Block for Accountability

Anupam Datta
Carnegie Mellon University
Email: danupam@cmu.edu

Deepak Garg
Max Planck Institute for Software Systems
Email: dg@mpi-sws.org

Dilsun Kaynar
Carnegie Mellon University
Email: dilsunk@cmu.edu

Divya Sharma
Carnegie Mellon University
Email: divyasharma@cmu.edu

Arunesh Sinha
University of Southern California
Email: aruneshs@usc.edu

***Abstract*—Protocols for tasks such as authentication, electronic voting, and secure multiparty computation ensure desirable security properties if agents follow their prescribed programs. However, if some agents deviate from their prescribed programs and a security property is violated, it is important to hold agents *accountable* by determining which deviations actually caused the violation. Motivated by these applications, we initiate a formal study of *program actions as actual causes*. Specifically, we define in an interacting program model what it means for a set of program actions to be an actual cause of a violation. We present a sound technique for establishing program actions as actual causes. We demonstrate the value of this formalism in two ways. First, we prove that violations of a specific class of safety properties always have an actual cause. Thus, our definition applies to relevant security properties. Second, we provide a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure.**

***Keywords*-Security Protocols, Accountability, Audit, Causation**

I. Introduction

Ensuring accountability for security violations is essential in a wide range of settings. For example, protocols for authentication and key exchange [1], electronic voting [2], auctions [3], and secure multiparty computation (in the semi-honest model) [4] ensure desirable security properties if protocol parties follow their prescribed programs. However, if they deviate from their prescribed programs and a security property is violated, determining which agents should be held accountable and appropriately punished is important to deter agents from committing future violations. Indeed the importance of accountability in information systems has been recognized in prior work [5], [6], [7], [8], [9], [10], [11]. Our thesis is that *actual causation* (i.e., identifying which agents' actions caused a specific violation) is a useful building block for accountability in decentralized multi-agent systems, including but not limited to security protocols and ceremonies [12].

Causation has been of interest to philosophers and ideas from philosophical literature have been introduced into computer science by the seminal work of Halpern and Pearl [13], [14], [15]. In particular, counterfactual reasoning is appealing as a basis for study of causation. Much of the definitional activity has centered around the question of what it means for event c to be an actual cause of event e. An answer to this question is useful to arrive at causal judgments for specific scenarios such as "John's smoking causes John's cancer" rather than general inferences such as "smoking causes cancer" (The latter form of judgments are studied in the related topic of type causation [15]). Notably, Hume [16] identified actual causation with counterfactual dependence—the idea that c is an actual cause of e if had c not occurred then e would not have occurred. While this simple idea does not work if there are independent causes, the counterfactual interpretation of actual causation has been developed further and formalized in a number of influential works (see, for example, [17], [15], [18], [19], [20], [14]).

Even though applications of counterfactual causal analysis are starting to emerge in the fields of AI, model-checking, and programming languages, causation has not yet been studied in connection with security protocols and violations thereof. On the other hand, causal analysis seems to be an intuitive building block for answering some very natural questions that have direct relevance to accountability such as (i) *why* a particular violation occurred, (ii) *what* component in the protocol is blameworthy for the violation and (iii) *how* the protocol could have been designed differently to preempt violations of this sort. Answering these questions requires an in-depth study of, respectively, explanations, blame-assignment, and protocol design, which are interesting problems in their own right, but are not the explicit focus of this paper. Instead, we focus on a formal definition of causation that we believe formal studies of these problems will need. Roughly speaking, explanations can be used to provide an *account* of the violation, *blame assignment* can be used to hold agents *accountable* for the violation, and protocol design informed by these would lead to protocols

with better accountability guarantees. We further elaborate on explanations and blame-assignment in Section V.

Formalizing actual causes as a building block for accountability in decentralized multi-agent systems raises new conceptual and technical challenges beyond those addressed in the literature on events as actual causes. In particular, prior work does not account for the program dynamics that arise in such settings. Let us consider a simple protocol example. In the movie *Flight* [21], a pilot drinks and snorts cocaine before flying a commercial plane, and the plane goes into a locked dive in mid-flight. While the pilot's behavior is found to be deviant in this case—he does not follow the prescribed protocol (program) for pilots—it is found to not be an actual cause of the plane's dive. The actual cause was a deviant behavior by the maintenance staff—they did not replace a mechanical component that should have been replaced. Ideally, the maintenance staff should have inspected the plane prior to take-off according to their prescribed protocol.

This example is useful to illustrate several key ideas that influence the formal development in this paper. First, it illustrates the importance of capturing the *actual interactions* among agents in a decentralized multi-agent system with non-deterministic execution semantics. The events in the movie could have unfolded in a different order but it is clear that the actual cause determination needs to be done based on the sequence of events that happened in reality. For example, had the maintenance staff replaced the faulty component *before* the take-off the plane would not have gone into a dive. Second, the example motivates us to hold accountable agents who exercise their choice to execute a deviant *program* that actually caused a violation. The maintenance staff had the choice to replace the faulty component or not where the task of replacing the component could consist of multiple steps. It is important to identify which of those steps were crucial for the occurrence of the dive. Thus, we focus on formalizing *program actions* executed in sequence (by agents) as actual causes of violations rather than individual, independent events as formalized in prior work. Finally, the example highlights the difference between deviance and actual causes—a difference also noted in prior work on actual causation. This difference is important from the standpoint of accountability. In particular, the punishment for deviating from the prescribed protocol could be suspension or license revocation whereas the punishment for actually causing a plane crash in which people died could be significantly higher (e.g., imprisonment for manslaughter). The first and second ideas, reflecting our program-based treatment, are the most significant points of difference from prior work on actual causation [14], [22] while the third idea is a significant point of difference from prior work in accountability [23], [24], [11], [7].

The central contribution of this paper is a formal definition of *program actions as actual causes*. Specifically, we define what it means for a set of program actions to be an actual cause of a violation. The definition considers a set of interacting programs whose concurrent execution, as recorded in a log, violates a trace property. It identifies a subset of actions (program steps) of these programs as an actual cause of the violation. The definition applies in two phases. The first phase identifies what we call *Lamport causes*. A Lamport cause is a minimal prefix of the log of a violating trace that can account for the violation. In the second phase, we refine the actions on this log by removing the actions which are merely *progress enablers* and obtain *actual action causes*. The former contribute only indirectly to the cause by enabling the actual action causes to make progress; the exact values returned by progress enabling actions are irrelevant.

We demonstrate the value of this formalism in two ways. First, we prove that violations of a precisely defined class of safety properties always have an actual cause. Thus, our definition applies to relevant security properties. Second, we provide a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure. Moreover, our example illustrates that our definition cleanly handles the separation between joint and independent causes –a recognized challenge for actual cause definitions [13], [14], [15].

In addition, we discuss how this formalism can serve as a building block for causal explanations and exoneration (i.e., soundly identifying agents who should not be blamed for a violation). We leave the technical development of these concepts for future work.

The rest of the paper is organized as follows. Section II describes a representative example which we use throughout the paper to explain important concepts. Section III gives formal definitions for program actions as actual causes of security violations. We apply the causal analysis to the running example in Section IV. We discuss the use of our causal analysis techniques for providing explanations and assigning blame in Section V. We survey additional related work in Section VI and conclude in Section VII.

II. Motivating example

In this section we describe an example protocol designed to increase accountability in the current public key infrastructure. We use the protocol later to illustrate key concepts in defining causality.

Security protocol: Consider an authentication protocol in which a user (User1) authenticates to a server (Server1) using a pre-shared password over an adversarial network. User1 sends its user-id to Server1 and obtains a public key signed by Server1. However, User1 would need inputs from additional sources when Server1 sends its public key for the first time in a protocol session to verify that the key is indeed bound to Server1's identity. In particular, User1 can verify the key by contacting multiple notaries

in the spirit of *Perspectives* [25]. For simplicity, we assume User1 verifies Server1's public key with three authorized notaries—Notary1, Notary2, Notary3—and accepts the key if and only if the majority of the notaries say that the key is legitimate. To illustrate some of our ideas, we also consider a parallel protocol where two parties (User2 and User3) communicate with each other.

We assume that the prescribed programs for Server1, User1, Notary1, Notary2, Notary3, User2 and User3 impose the following requirements on their behavior: (i) Server1 stores User1's password in a hashed form in a secure private memory location. (ii) User1 requests access to the account by sending an encryption of the password (along with its identity and a timestamp) to Server1 after verifying Server1's public key with a majority of the notaries. (iii) The notaries retrieve the key from their databases and attest the key correctly. (iv) Server1 decrypts and computes the hashed value of the password. (v) Server1 matches the computed hash value with the previously stored value in the memory location when the account was first created; if the two hash values match, then Server1 grants access to the account to User1. (vi) In parallel, User2 generates and sends a nonce to User3. (vii) User3 generates a nonce and responds to User2.

Security property: The prescribed programs in our example aim to achieve the property that only the user who created the account and password (in this case, User1) gains access to the account.

Compromised Notaries Attack: We describe an attack scenario and use it to illustrate nuances in formalizing program actions as actual causes. User1 executes its prescribed program. User1 sends an access request to Server1. An Adversary intercepts the message and sends a public key to User1 pretending to be Server1. User1 checks with Notary1, Notary2 and Notary3 who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to User1's account. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 forwards the message to User3. We assume that the actions of the parties are recorded on a *log*, say l. Note that this log contains a violation of the security property described above since Adversary gains access to an account owned by User1.

First, our definition finds *program actions as causes* of violations. At a high-level, as mentioned in the introduction, our definition applies in two phases. The first phase (Section III, Definition 12) identifies a minimal prefix (Phase 1, *minimality*) of the log that can account for the violation i.e. we consider all scenarios where the sequence of actions execute in the same order as on the log, and test whether it suffices to recreate the violation in the absence of all other actions (Phase 1, *sufficiency*). In our example, this first phase will output a minimal prefix of log l above. In this case, the minimal prefix will not contain interactions between User2 and User3 after Server1 has granted access to the Adversary (the remaining prefix will still contain a violation).

Second, a nuance in defining the notion of *sufficiency* (Phase 1, Definition 12) is to constrain the interactions which are a part of the actual cause set in a manner that is consistent with the interaction recorded on the log. This constraint on interactions is quite subtle to define and depends on how strong a coupling we find appropriate between the log and possible counterfactual traces in sufficiency: if the constraint is too weak then the violation does not reappear in all sequences, thus missing certain causes; if it is too strong it leads to counter-intuitive cause determinations. For example, a weak notion of consistency is to require that each program locally execute the same prefix in sufficiency as it does on the log i.e. consistency w.r.t. program actions for individual programs. This notion does not work because for some violations to occur the *order of interactions* on the log among programs is important. A notion that is too strong is to require matching of the total order of execution of all actions across all programs. We present a formal notion of *consistency* by comparing log projections (Section III-B) that balance these competing concerns.

Third, note that while Phase 1 captures a minimal prefix of the log sufficient for the violation, it might be possible to remove actions from this prefix which are merely required for a program execution to progress. For instance note that while all three notaries' actions are required for User1 to progress (otherwise it would be stuck waiting to receive a message) and the violation to occur, the actual message sent by one of the notaries is irrelevant since it does not affect the majority decision in this example. Thus, separating out actions which are *progress enablers* from those which provide information that causes the violation is useful for fine-grained causal determination. This observation motivates the final piece (Phase 2) of our formal definition (Definition 14).

Finally, notice that in this example Adversary, Notary1, Notary2, Notary3, Server1 and User2 deviate from the protocol described above. However, the deviant programs are not sufficient for the violation to occur without the involvement of User1, which is also a part of the causal set. We thus seek a notion of sufficiency in defining a set of programs as a joint actual cause for the violation. Joint causation is also significant in legal contexts [26]. For instance, it is useful for holding liable a group of agents working together when none of them satisfy the cause criteria individually but together their actions are found to be a cause. The ability to distinguish between joint and independent (i.e., different sets of programs that independently caused the violation) causes is an important criterion that we want our definition to satisfy. In particular, Phase 2 of our definition helps identify independent causes. For instance, in our example, we get three different independent causes depending on which notary's action is treated as a progress enabler. Our ultimate goal is to use the notion of actual cause as a building

block for accountability — the independent vs. joint cause distinction is significant when making deliberations about accountability and punishment for liable parties. We can use the result of our causal determinations to further remove deviants whose actions are required for the violation to occur but might not be blameworthy (Section V).

III. Actual Cause Definition

We present our language model in Section III-A, auxiliary notions in Section III-B, properties of interest to our analysis in Section III-C, and the formal definition of program actions as actual causes in Section III-D.

A. Model

We model programs in a simple concurrent language, which we call L. The language contains sequential expressions, e, that execute concurrently in threads and communicate with each other through `send` and `recv` commands. Terms, t, denote messages that may be passed through expressions or across threads. Variables x range over terms. An expression is a sequence of actions, α. An action may do one of the following: execute a primitive function ζ on a term t (written $\zeta(t)$), or send or receive a message to another thread (written $\texttt{send}(t)$ and $\texttt{recv}()$, respectively). We also include very primitive condition checking in the form of $\texttt{assert}(t)$.

Terms	t	::=	$x \mid \ldots$
Actions	α	::=	$\zeta(t) \mid \texttt{send}(t) \mid \texttt{recv}()$
Expressions	e	::=	$t \mid (b : x = \alpha); e_2 \mid \texttt{assert}(t); e$

Each action α is labeled with a unique line number, written b. Line numbers help define traces later. We omit line numbers when they are irrelevant. Every action and expression in the language evaluates to a term and potentially has side-effects. The term returned by action α is bound to x in evaluating e_2 in the expression $(b : x = \alpha); e_2$.

Following standard models of protocols, `send` and `recv` are untargeted in the operational semantics: A message sent by a thread may be received by any thread. Targeted communication may be layered on this basic semantics using cryptography. For readability in examples, we provide an additional first argument to `send` and `recv` that specifies the *intended* target (the operational semantics ignore this intended target). Action $\texttt{send}(t)$ always returns 0 to its continuation.

Primitive functions ζ model thread-local computation like arithmetic and cryptographic operations. Primitive functions can also read and update a *thread-local state*, which may model local databases, permission matrices, session information, etc. If the term t in $\texttt{assert}(t)$ evaluates to a non-true value, then its containing thread gets stuck forever, else $\texttt{assert}(t)$ has no effect.

We abbreviate $(b : x = \alpha); x$ to $b : \alpha$ and $(b : x = \alpha); e$ to $(b : \alpha); e$ when x is not free in e. As an example, the following expression receives a message, generates a nonce (through a primitive function `new`) and sends the concatenation of the received message and the nonce on the network to the intended recipient j (line numbers are omitted here).

$m = \texttt{recv}();$ //receive message, bind to m
$n = \texttt{new}();$ //generate nonce, bind to n
$\texttt{send}(j, (m, n));$ //send (m, n) to j

For the purpose of this paper, we limit attention to this simple expression language, without recursion or branching. Our definition of actual cause is general and applies to any formalism of (non-deterministic) interacting agents, but the auxiliary definitions of log projection and the function dummify introduced later must be modified.

Operational Semantics: The language L's operational semantics define how a collection of *threads* execute concurrently. Each thread T contains a unique thread identifier i (drawn from a universal set of such identifiers), the executing expression e, and a local store. A *configuration* $\mathcal{C} = T_1, \ldots, T_n$ models the threads $T_1, \ldots, T_n$ executing concurrently. Our reduction relation is written $\mathcal{C} \to \mathcal{C}'$ and defined in the standard way by interleaving small steps of individual threads (the reduction relation is parametrized by a semantics of primitive functions ζ). Importantly, each reduction can either be internal to a single thread or a *synchronization* of a `send` in one thread with a `recv` in another thread.

We make the locus of a reduction explicit by annotating the reduction arrow with a *label* r. This is written $\mathcal{C} \xrightarrow{r} \mathcal{C}'$. A label is either the identifier of a thread i paired with a line number b, written $\langle i, b\rangle$ and representing an internal reduction of some $\zeta(t)$ in thread i at line number b, or a tuple $\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle$, representing a synchronization between a `send` at line number b_s in thread i_s with a `recv` at line number b_r in thread i_r, or ϵ indicating an unobservable reduction (of t or $\texttt{assert}(t)$) in some thread. Labels $\langle i, b\rangle$ are called *local labels*, labels $\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle$ are called *synchronization labels* and labels ϵ are called *silent labels*.

An *initial configuration* can be described by a triple $\langle I, \mathcal{A}, \Sigma\rangle$, where I is a finite set of thread identifiers, $\mathcal{A} : I \to$ Expressions and $\Sigma : I \to$ Stores. This defines an initial configuration of $|I|$ threads with identifiers in I, where thread i contains the expression $\mathcal{A}(i)$ and the store $\Sigma(i)$. In the sequel, we identify the triple $\langle I, \mathcal{A}, \Sigma\rangle$ with the configuration defined by it. We also use a configuration's identifiers to refer to its threads.

Definition 1 (Run): Given an initial configuration $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$, a run is a finite sequence of labeled reductions $\mathcal{C}_0 \xrightarrow{r_1} \mathcal{C}_1 \ldots \xrightarrow{r_n} \mathcal{C}_n$.

A pre-trace is obtained by projecting only the stores from each configuration in a run.

Definition 2 (Pre-trace): Let $\mathcal{C}_0 \xrightarrow{r_1} \mathcal{C}_1 \ldots \xrightarrow{r_n} \mathcal{C}_n$ be a run and let Σ_i be the store in configuration

$\mathcal{C}_i$. Then, the pre-trace of the run is the sequence $(\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$.

If $r_i = \epsilon$, then the ith step is an unobservable reduction in some thread and, additionally, $\Sigma_{i-1} = \Sigma_i$. A trace is a pre-trace from which such ϵ steps have been dropped.

Definition 3 (Trace): The trace of the pre-trace $(\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$ is the subsequence obtained by dropping all tuples of the form (ϵ, Σ_i). Traces are denoted with the letter t.

B. Logs and their projections

To define actual causation, we find it convenient to introduce the notion of a log and the log of a trace, which is just the sequence of non-silent labels on the trace. A log is a sequence of labels other than ϵ. The letter l denotes logs.

Definition 4 (Log): Given a trace $t = (\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$, the log of the trace, $log(t)$, is the sequence of $r_1, \ldots, r_m$. (The trace t does not contain a label r_i that equals ϵ, so neither does $log(t)$.)

We need a few more straightforward definitions on logs in order to define actual causation.

Definition 5 (Projection of a log): Given a log l and a thread identifier i, the projection of l to i, written $l|_i$ is the subsequence of all labels in l that mention i. Formally,

$$
\begin{array}{ll}
\bullet|_i & = \bullet \\
(\langle i, b\rangle :: l)|_i & = \langle i, b\rangle :: (l|_i) \\
(\langle i, b\rangle :: l)|_i & = l|_i \qquad \text{if } i \neq j \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_i & = \langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: (l|_i) \\
 & \quad \text{if } i_s = i \text{ or } i_r = i \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_i & = l|_i \\
 & \quad \text{if } i_s \neq i \text{ and } i_r \neq i
\end{array}
$$

Definition 6 (Projected prefix): We call a log l' a *projected prefix* of the log l, written $l' \leq_p l$, if for every thread identifier i, the sequence $l'|_i$ is a prefix of the sequence $l|_i$.

The definition of projected prefix allows the relative order of events in two different non-communicating threads to differ in l and l' but Lamport's happens-before order of actions [27] in l' must be preserved in l. Similar to projected prefix, we define projected sublog.

Definition 7 (Projected sublog): We call a log l' a *projected sublog* of the log l, written $l' \sqsubseteq_p l$, if for every thread identifier i, the sequence $l'|_i$ is a subsequence of the sequence $l|_i$ (i.e., dropping some labels from $l|_i$ results in $l'|_i$).

C. Properties of Interest

A *property* is a set of (good) traces and violations are traces in the complement of the set. Our goal is to define the cause of a violation of a property. We are specifically interested in ascribing causes to violations of safety properties [28] because safety properties encompass many relevant security requirements. We recapitulate the definition of a safety property below. Briefly, a property is safety if it is fully characterized by a set of finite violating prefixes of traces. Let U denote the universe of all possible traces.

Definition 8 (Safety property [29]): A property P (a set of traces) is a safety property, written $\mathsf{Safety}(P)$, if $\forall t \notin P.\ \exists t' \in U.\ (t'$ is a prefix of $t) \wedge (\forall t'' \in U.\ (t' \cdot t'' \notin P))$.

As we explain soon, our causal analysis ascribes thread actions (or threads) as causes. One important requirement for such analysis is that the property be closed under reordering of actions in different threads if those actions are not related by Lamport's happens-before relation [27]. For properties that are not closed in this sense, the *global order* between actions in a race condition may be a cause of a violation. Whereas causal analysis of race conditions may be practically relevant in some situation, we limit attention only to properties that are closed in the sense described here. We call such properties reordering-closed or RC.

Definition 9 (Reordering-equivalence): Two traces t_1, t_2 starting from the same initial configuration are called reordering-equivalent, written $t_1 \sim t_2$ if for each thread identifier i, $log(t_1)|_i = log(t_2)|_i$. Note that $\sim$ is an equivalence relation on traces from a given initial configuration. Let $[t]_\sim$ denote the equivalence class of t.

Definition 10 (Reordering-closed property): A property P is called reordering-closed, written $\mathsf{RC}(P)$, if $t \in P$ implies $[t]_\sim \subseteq P$. Note that $\mathsf{RC}(P)$ iff $\mathsf{RC}(\neg P)$.

D. Program Actions as Actual Causes

In the sequel, let φ_V denote the *complement* of a reordering-closed safety property of interest. (The subscript V stands for "violations".) Consider a trace t starting from the initial configuration $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$. If $t \in \varphi_V$, then t violates the property $\neg\varphi_V$.

Definition 11 (Violation): A violation of the property $\neg\varphi_V$ is a trace $t \in \varphi_V$.

Our definition of actual causation identifies a subset of actions in $\{\mathcal{A}(i) \mid i \in I\}$ as the cause of a violation $t \in \varphi_V$. The definition applies in two phases. The first phase identifies what we call *Lamport causes*. A Lamport cause is a minimal projected prefix of the log of a violating trace that can account for the violation. In the second phase, we refine the log by removing actions that are merely *progress enablers*; the remaining actions on the log are the *actual action causes*. The former contribute only indirectly to the cause by enabling the actual action causes to make progress; the exact values returned by progress enabling actions are irrelevant.

The following definition, called Phase 1, determines Lamport causes. It works as follows. We first identify a projected prefix l of the log of a violating trace t as a potential candidate for a Lamport cause. We then check two conditions on l. The *sufficiency* condition tests that the threads of the configuration, when executed at least up to the identified prefix, preserving all synchronizations in the prefix, suffice to recreate the violation. The *minimality*

condition tests that the identified Lamport cause contains no redundant actions.

Definition 12 (Phase 1: Lamport Cause of Violation): Let $t \in \varphi_V$ be a trace starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma \rangle$ and l be a projected prefix of $log(t)$, i.e., $l \leq_p log(t)$. We say that l is the Lamport cause of the violation t of φ_V if the following hold:

1) **(Sufficiency)** Let T be the set of traces starting from $\mathcal{C}_0$ whose logs contain l as a projected prefix, i.e., $T = \{t' \mid t' \text{ is a trace starting from } \mathcal{C}_0 \text{ and } l \leq_p log(t')\}$. Then, every trace in T has the violation φ_V, i.e., $T \subseteq \varphi_V$. (Because $t \in T$, T is non-empty.)
2) **(Minimality)** No proper prefix of l satisfies condition 1.

At the end of Phase 1, we obtain one or more minimal prefixes l which contain program actions that are sufficient for the violation. These prefixes represent independent Lamport causes of the violation. In the Phase 2 definition below, we further identify a sublog a_d of each l, such that the program actions in a_d are actual causes and the actions in $l \backslash a_d$ are progress enabling actions which only contribute towards the *progress* of actions in a_d that cause the violation. In other words, the actions not considered in a_d contain all labels whose actual returned values are irrelevant.

Briefly, here's how our Phase 2 definition works. We first pick a candidate projected sublog a_d of l, where log l is a Lamport cause identified in Phase 1. We consider counterfactual traces obtained from initial configurations in which program actions omitted from a_d are replaced by actions that do not have any effect other than enabling the program to progress (referred to as no-op). If a violation appears in all such counterfactual traces, then this sublog a_d is a good candidate. Of all such good candidates, we choose those that are minimal.

The key technical difficulty in writing this definition is replacing program actions omitted from a_d with no-ops. We cannot simply erase any such action because the action is expected to return a term which is bound to a variable used in the action's continuation. Hence, our approach is to substitute the variables binding the returns of no-op'ed actions with arbitrary (side-effect free) terms t. Formally, we assume a function $f : I \times \text{LineNumbers} \to \text{Terms}$ that for line number b in thread i suggests a suitable term $f(i, b)$ that must be returned if the action from line b in thread i is replaced with a no-op. In our cause definition we universally quantify over f, thus obtaining the effect of a no-op. For technical convenience, we define a syntactic transform called $\mathsf{dummify}()$ that takes an initial configuration, the chosen sublog a_d and the function f, and produces a new initial configuration obtained by erasing actions not in a_d by terms obtained through f.

Definition 13 (Dummifying transformation): Let $\langle I, \mathcal{A}, \Sigma \rangle$ be a configuration and let a_d be a log. Let $f : I \times \text{LineNumbers} \to \text{Terms}$. The dummifying transform $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ is the initial configuration $\langle I, \mathcal{D}, \Sigma \rangle$, where for all $i \in I$, $\mathcal{D}(i)$ is $\mathcal{A}(i)$ modified as follows:

- If $(b : x = \texttt{send}(t)); e$ appears in $\mathcal{A}(i)$ but $\langle i, b \rangle$ does not appear in a_d, then replace $(b : x = \texttt{send}(t)); e$ with $e[0/x]$ in $\mathcal{A}(i)$.
- If $(b : x = \alpha); e$ appears in $\mathcal{A}(i)$ but $\langle i, b \rangle$ does not appear in a_d and $\alpha \neq \texttt{send}(\_)$, then replace $(b : x = \alpha); e$ with $e[f(i, b)/x]$ in $\mathcal{A}(i)$.

We now present our main definition of actual causes.

Definition 14 (Phase 2: Actual Cause of Violation): Let $t \in \varphi_V$ be a trace from the initial configuration $\langle I, \mathcal{A}, \Sigma \rangle$ and let the log $l \leq_p log(t)$ be a Lamport cause of the violation determined by Definition 12. Let a_d be a projected sublog of l, i.e., let $a_d \sqsubseteq_p l$. We say that a_d is the actual cause of violation t of φ_V if the following hold:

1) **(Sufficiency')** Pick any f. Let $\mathcal{C}'_0 = \mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ and let T be the set of traces starting from $\mathcal{C}'_0$ whose logs contain a_d as a projected sublog, i.e., $T = \{t' \mid t' \text{ is a trace starting from } \mathcal{C}'_0 \text{ and } a_d \sqsubseteq_p log(t')\}$. Then, T is non-empty and every trace in T has the violation φ_V, i.e, $T \subseteq \varphi_V$.
2) **(Minimality')** No proper sublog of a_d satisfies condition 1.

At the end of Phase 2, we obtain one or more sets of actions a_d. These sets are deemed the independent actual causes of the violation t.

The following theorem states that for all safety properties that are re-ordering closed, the Phase 1 and Phase 2 definitions always identify at least one Lamport and at least one actual cause.

Theorem 1: Suppose φ_V is reordering-closed and the complement of a safety property, i.e., $\mathsf{RC}(\varphi_V)$ and $\mathsf{safety}(\neg\varphi_V)$. Then, for every $t \in \varphi_V$: (1) Our Phase 1 definition (Definition 12) finds a Lamport cause l, and (2) For every such Lamport cause l, the Phase 2 definition (Definition 14) finds an actual cause a_d.

Proof: (1) Pick any $t \in \varphi_V$. We follow the Phase 1 definition. It suffices to prove that there is a log $l \leq_p log(t)$ that satisfies the sufficiency condition. Since $\mathsf{safety}(\neg\varphi_V)$, there is a prefix t_0 of t s.t. for all $t_1 \in U$, $t_0 \cdot t_1 \in \varphi_V$. Choose $l = log(t_0)$. Since t_0 is a prefix of t, $l = log(t_0) \leq_p log(t)$. To prove sufficiency, pick any trace t' s.t. $l \leq_p log(t')$. It suffices to prove $t' \in \varphi_V$. Since $l \leq_p log(t')$, for each i, $log(t')|_i = l|_i \cdot l'_i$ for some l'_i. Let t'' be the (unique) subsequence of t' containing all labels from the logs $\{l'_i\}$. Consider the trace $s = t_0 \cdot t''$. First, s extends t_0, so $s \in \varphi_V$. Second, $s \sim t'$ because $log(s)|_i = l|_i \cdot l'_i = log(t_0)|_i \cdot log(t'')|_i = log(t_0 \cdot t'')|_i = log(t')|_i$. Since $\mathsf{RC}(\varphi_V)$, $t' \in \varphi_V$.

(2) Pick any $t \in \varphi_V$ and let l be a Lamport cause of t as determined by the Phase 1 definition. Following the Phase 2 definition, we only need to prove that there is at

least one $a_d \sqsubseteq_p l$ that satisfies the sufficiency' condition. We choose $a_d = l$. To show sufficiency', pick any f. Because $a_d = l$, a_d specifies an initial prefix of every $\mathcal{A}(i)$ and the transform $\mathsf{dummify}()$ has no effect on this prefix. First, we need to show that at least one trace t' starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ satisfies $a_d \sqsubseteq_p log(t')$. For this, we can pick $t' = t$. Second, we need to prove that any trace t' starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ s.t. $a_d \sqsubseteq_p log(t')$ satisfies $t' \in \varphi_V$. Pick such a t'. Let t_0 be the prefix of t corresponding to l. Then, $log(t_0)|_i = l|_i$ for each i. It follows immediately that for each i, $t'|_i = t_0|_i \cdot t''_i$ for some t''_i. Let t'' be the unique subsequence of t' containing all labels from traces $\{t''_i\}$. Let $s = t_0 \cdot t''$. First, because for each i, $l|_i = log(t_0)|_i$, $l \leq_p log(t_0)$ trivially. Because l is a Lamport cause, it satisfies the sufficiency condition of Phase 1, so $t_0 \in \varphi_V$. Since $\mathsf{safety}(\neg\varphi_V)$, and s extends t_0, $s \in \varphi_V$. Second, $s \sim t'$ because $log(s)|_i = log(t_0)|_i \cdot log(t'')|_i = log(t')|_i$ and both s and t' are traces starting from the initial configuration $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$. Hence, by $\mathsf{RC}(\varphi_V)$, $t' \in \varphi_V$. ■

Our Phase 2 definition identifies a set of program actions as causes of a violation. However, in some applications it may be necessary to ascribe thread identifiers (or programs) as causes. This can be straightforwardly handled by lifting the Phase 2 definition: A thread i (or $\mathcal{A}(i)$) is a cause if one of its actions appears in a_d.

Definition 15 (Program Cause of Violation): Let a_d be an actual cause of violation φ_V on trace t starting from $\langle I, \mathcal{A}, \Sigma\rangle$. We say that the set $X \subseteq I$ of thread identifiers is a cause of the violation if $X = \{i \mid i \text{ appears in } a_d\}$.

Remarks: We make a few technical observations about our definitions of cause. First, because Lamport causes (Definition 12) are projected *prefixes*, they contain all actions that occur before any action that actually contributes to the violation. Many of the actions in the Lamport cause may not contribute to the violation intuitively. Our actual cause definition filters out such "spurious" actions. As an example, suppose that a safety property requires that the value 1 never be sent on the network. The (only) trace of the program $x = 1; y = 2; z = 3; \mathtt{send}(x)$ violates this property. The Lamport cause of this violation contains all four actions of the program, but it is intuitively clear that the two actions $y = 2$ and $z = 3$ do not contribute to the violation. Indeed, the actual cause of the violation determined by Definition 14 does not contain these two actions; it contains only $x = 1$ and $\mathtt{send}(x)$, both of which obviously contribute to the violation.

Second, our definition of dummification is based on a program transformation that needs line numbers. One possibly unwanted consequence is that our traces have line numbers and, hence, we could, in principle, specify safety properties that are sensitive to line numbers. However, our definitions of cause are closed under bijective renaming of line numbers, so if a safety property is insensitive to line numbers, the actual causes can be quotiented under bijective renamings of line numbers.

Third, our definition of actual cause (Definition 14) separates actions whose return values are relevant to the violation from those whose return values are irrelevant for the violation. This is closely related to noninterference-like security definitions for information flow control, in particular, those that separate input presence from input content [30]. Lamport causes (Definition 12) have a trivial connection to information flow: If an action does not occur in any Lamport cause of a violation, then there cannot be an information flow from that action to the occurrence of the violation.

IV. Causes of Authentication Failures

In this section, we model an instance of our running example based on passwords (Section II) in order to demonstrate our actual cause definition. As explained in Section II, we consider a protocol session where Server1, User1, User2, User3 and multiple notaries interact over an adversarial network to establish access over a password-protected account. We describe a formal model of the protocol in our language, examine the attack scenario from Section II and provide a cause analysis using the definitions from Section III.

A. Protocol Description

We consider our example protocol with eight threads named {Server1, User1, Adversary, Notary1, Notary2, Notary3, User2, User3}. In this section, we briefly describe the protocol and the programs specified by the protocol for each of these threads. For this purpose, we assume that we are provided a function $\mathcal{N} : I \to$ Expressions such that $\mathcal{N}(i)$ is the program that *ideally should have been* executing in the thread i. For each i, we call $\mathcal{N}(i)$ the *norm* for thread i. The violation is caused because some of the executing programs are different from the norms. These actual programs, called $\mathcal{A}$ as in Section III, are shown later. The norms are shown here to help the reader understand what the ideal protocol is and also to facilitate some of the development in Section V. The appendix describes an expansion of this example with more than the eight threads considered here to illustrate our definitions better. The proof included in the appendix deals with timestamps and signatures.

The norms in Figure 1 and the actuals in Figure 2 assume that User1's account (called $acct$ in Server1's program) has already been created and that User1's password, pwd is associated with User1's user id, uid. This association (in hashed form) is stored in Server1's local state at pointer mem. The norm for Server1 is to wait for a request from an entity, respond with its (Server1's) public key, wait for a username-password pair encrypted with that public key and grant access to the requester if the password matches the previously stored value in Server1's memory at mem. To grant access, Server1 adds an entry into a private access

matrix, called P. (A separate server thread, not shown here, allows User1 to access its account if this entry exists in P.)

The norm for User1 is to send an access request to Server1, wait for the server's public key, verify that key with three notaries and then send its password pwd to Server1, encrypted under Server1's public key. On receiving Server1's public key, User1 initiates a protocol with the three notaries and accepts or rejects the key based on the response of a majority of the notaries. For simplicity, we omit a detailed description of this protocol between User1 and the notaries that authenticates the notaries and ensures freshness of their responses. These details are included in our appendix. In parallel, the norm for User2 is to generate and send a nonce to User3. The norm for User3 is to receive a message from User2, generate a nonce and send it to User2.

Each notary has a private database of *(public_key, principal)* tuples. The notaries' norms assume that this database has already been created correctly. When User1 sends a request with a public key, the notary responds with the principal's identifier after retrieving the tuple corresponding to the key from its database.

Notation: The programs in this example use several primitive functions ζ. $\texttt{Enc}(k, m)$ and $\texttt{Dec}(k', m)$ denote encryption and decryption of message m with key k and k' respectively. $\texttt{Hash}(m)$ generates the hash of term m. $\texttt{Sig}(k, m)$ denotes message m signed with the key k, paired with m in the clear. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For readability, we include the intended recipient i and expected sender j of a message as the first argument of $\texttt{send}(i, m)$ and $\texttt{recv}(j)$ expressions. As explained earlier, i and j are ignored during execution and a network adversary, if present, may capture or inject any messages.

Security property: The security property of interest to us is that if at time u, a thread k is given access to account a, then k owns a. Specifically, in this example, we are interested in case $a = acct$ and $k =$ User1. This can be formalized by the following logical formula, $\neg\varphi_V$:

$$\forall u, k.\ (acct, k) \in P(u) \supset (k = \mathsf{User1}) \qquad (1)$$

Here, $P(u)$ is the state of the access control matrix P for Server1 at time u.

B. Attack

As an illustration, we model the "Compromised Notaries" violation of Section II. The programs executed by all threads are given in Figure 2. User1 sends an access request to Server1 which is intercepted by Adversary who sends its own key to User1 (pretending to be Server1). User1 checks with the three notaries who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to the User1's account. Note that the actual programs of the three notaries attest that the

Norm $\mathcal{N}$(Server1):

1 : _ = recv(j); //access req from thread j
2 : send(j, pub_key_Server1); //send public key to j
3 : s = recv(j); //encrypted uid, pwd, thread id J
4 : (uid, pwd, J) = Dec(pvt_key_Server1, s);
5 : t = hash(uid, pwd);
assert($mem = t$) //compare hash with stored value
6 : insert(P, ($acct$, J));

Norm $\mathcal{N}$(User1):

1 : send(Server1); //access request
2 : pub_key = recv(Server1); //key from Server1
3 : send(Notary1, pub_key);
4 : send(Notary2, pub_key);
5 : send(Notary3, pub_key);
6 : Sig(pub_key, $l1$) = recv(Notary1); //notary1 responds
7 : Sig(pub_key, $l2$) = recv(Notary2); //notary2 responds
8 : Sig(pub_key, $l3$) = recv(Notary3); //notary3 responds
assert(At least two of {l1,l2,l3} equal Server1)
9 : t = Enc(pub_key, (uid, pwd, User1));
10 : send(Server1, t); //send t to Server1

Norms $\mathcal{N}$(Notary1), $\mathcal{N}$(Notary2), $\mathcal{N}$(Notary3):

// o denotes Notary1, Notary2 or Notary3
1 : pub_key = recv(j);
2 : pr = KeyOwner(pub_key); //lookup key owner
3 : send(j, Sig(pvt_key_o, (pub_key, pr));

Norm $\mathcal{N}$(User2):

1 : send(User3);
2 : _ = recv(User3);

Norm $\mathcal{N}$(User3):

1 : _ = recv(User2);
2 : send(User3);

Figure 1. Norms for all threads. Adversary's norm is the trivial empty program.

public key given to them belongs to Server1. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 interacts with User3, as in their norms.

Figure 3 shows the expressions executed by each thread on the property-violating trace. For instance, the label $\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle$ indicates that both User1 and Adversary executed the expressions with the line number 1 in their actual programs, which resulted in a synchronous communication between them, while the label $\langle \mathsf{Adversary}, 4\rangle$ indicates the local execution of the expression at line 4 of Adversary's program. The initial configuration has the programs: $\{\mathcal{A}(\mathsf{User1}), \mathcal{A}(\mathsf{Server1}), \mathcal{A}(\mathsf{Adversary}), \mathcal{A}(\mathsf{Notary1}),$ $\mathcal{A}(\mathsf{Notary2}), \mathcal{A}(\mathsf{Notary3}), \mathcal{A}(\mathsf{User2}), \mathcal{A}(\mathsf{User3})\}$. For this attack scenario, the concrete trace t we consider is such that $\log(t)$ is any *arbitrary interleaving* of the actions for $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{User2}, \mathsf{User3}, \mathsf{Server1}, \mathsf{Notary1},$ $\mathsf{Notary2}, \mathsf{Notary3}\}$ shown in Figure 3(a). Any such interleaved log is denoted $\log(t)$ in the sequel. At the end of this log, $(acct, \mathsf{Adversary})$ occurs in the access control

Actual $\mathcal{A}$(Server1)**:**

1 : _ = recv(j); //access req from thread j
2 : send(j, pub_key_Server1); //send public key to j
3 : _ = recv(j); //receive nonce from thread User2
4 : send(j); //send signed nonce
5 : s = recv(j); //encrypted uid, pwd, thread id from j
6 : (uid, pwd, J) = Dec(pvt_key_Server1, s);
7 : t = hash(uid, pwd);
assert($mem = t$)**[A]** //compare hash with stored value
8 : insert(P, $(acct, J)$);

Actual $\mathcal{A}$(User1)**:**

1 : send(Server1); //access request
2 : pub_key = recv(Server1); //key from Server1
3 : send(Notary1, pub_key);
4 : send(Notary2, pub_key);
5 : send(Notary3, pub_key);
6 : Sig($pub\_key, l1$) = recv(Notary1); //notary1 responds
7 : Sig($pub\_key, l2$) = recv(Notary2); //notary2 responds
8 : Sig($pub\_key, l3$) = recv(Notary3); //notary3 responds
assert(At least two of {l1,l2,l3} equal Server1)**[B]**
9 : t = Enc(pub_key, (uid, pwd, User1));
10 : send(Server1, t); //send t to Server1

Actual $\mathcal{A}$(Adversary)

1 : recv(User1); //intercept access req from User1
2 : send(User1, pub_key_A); //send key to User
3 : s = recv(User1); //pwd from User1
4 : $(uid, pwd,$ User1) = Dec(pvt_key_A, s); //decrypt pwd
5 : send(Server1, uid); //access request to Server1
6 : pub_key = recv(Server1); //Receive Server1's public key
7 : t = Enc(pub_key, (uid, pwd, Adversary)); //encrypt pwd
8 : send(Server1, t); //pwd to Server1

Actuals $\mathcal{A}$(Notary1), $\mathcal{A}$(Notary2), $\mathcal{N}$(Notary3)**:**

// o denotes Notary1, Notary2 or Notary3
1 : pub_key = recv(j);
2 : send(j, Sig(pvt_key_o, (pub_key, Server1)));

Actual $\mathcal{A}$(User2)**:**

1 : send(Server1); //send nonce to Server1
2 : _ = recv(Server1);
3 : send(User3); //forward nonce to User3
4 : _ = recv(User3);

Actual $\mathcal{A}$(User3)**:**

1 : _ = recv(User2);
2 : send(User2); //send nonce to User2

Figure 2. Actuals for all threads.

matrix P, but Adversary does not own $acct$. Hence, this log corresponds to a violation of our security property.

Note that if any two of the three notaries had attested the Adversary's key to belong to Server1, the violation would still have happened. Consequently, we may expect three independent program causes in this example: {Adversary, User1, Server1, Notary1, Notary2} with the action causes a_d as shown in Figure 3(c), {Adversary, User1, Server1, Notary1, Notary3} with the actions a'_d, and {Adversary, User1, Server1, Notary2, Notary3} with the actions a''_d where a'_d and a''_d can be obtained from a_d (Figure 3) by considering actions for {Notary1, Notary3} and {Notary2, Notary3} respectively, instead of actions for {Notary1, Notary2}. Our treatment of independent causes follows the tradition in the causality literature. The following theorem states that our definitions determine exactly these three independent causes – one notary is dropped from each of these sets, but no notary is discharged from all the sets. This determination reflects the intuition that only two dishonest notaries are sufficient to cause the violation. Additionally, while it is true that all parties who follow the protocol should not be *blamed* for a violation, an honest party may be an *actual cause* of the violation (in both the common and the philosophical sense of the word), as demonstrated in this case study. This two-tiered view of accountability of an action by separately asserting cause and blame can also be found in prior work in law and philosophy [5], [31]. Determining actual cause is nontrivial and is the focus of this work.

Theorem 2: Let $I = \{$User1, Server1, Adversary, Notary1, Notary2, Notary3, User2, User3$\}$ and Σ and $\mathcal{A}$ be as described above. Let t be a trace from $\langle I, \mathcal{A}, \Sigma\rangle$ such that $log(t)|_i$ for each $i \in I$ matches the corresponding log projection from Figure 3(a). Then, Definition 15 determines three possible values for the program cause X of violation $t \in \varphi_V$: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3} where the corresponding actual causes are a_d, a'_d and a''_d, respectively.

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that our Phase 1, Phase 2 definitions (Definitions 12, 14, 15) yield exactly the three values for X mentioned in the theorem.

Lamport cause (Phase 1): We show that any l whose projections match those shown in Figure 3(b) satisfies sufficiency and minimality. From Figure 3(b), such an l has no actions for User3 and only those actions of User2 that are involved in synchronization with Server1. For all other threads, the log contains every action from t. The intuitive explanation for this l is straightforward: Since l must be a (projected) *prefix* of the trace, and the violation only happens because of insert in the last statement of Server1's program, every action of every program before that statement in Lamport's happens-before relation must be in l. This is exactly the l described in Figure 3(b).

Formally, following the statement of sufficiency, let T be the set of traces starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$ (Figure 2) whose logs contain l as a projected prefix. Pick any $t' \in T$. We need to show $t' \in \varphi_V$. However, note that any t' containing all actions in l must also add $(acct,$ Adversary$)$ to P, but Adversary $\neq$ User1. Hence, $t' \in \varphi_V$. Further, l is minimal as described in the previous paragraph.

Actual cause (Phase 2): Phase 2 (Definitions 14, 15) determines three independent program causes for X: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3},

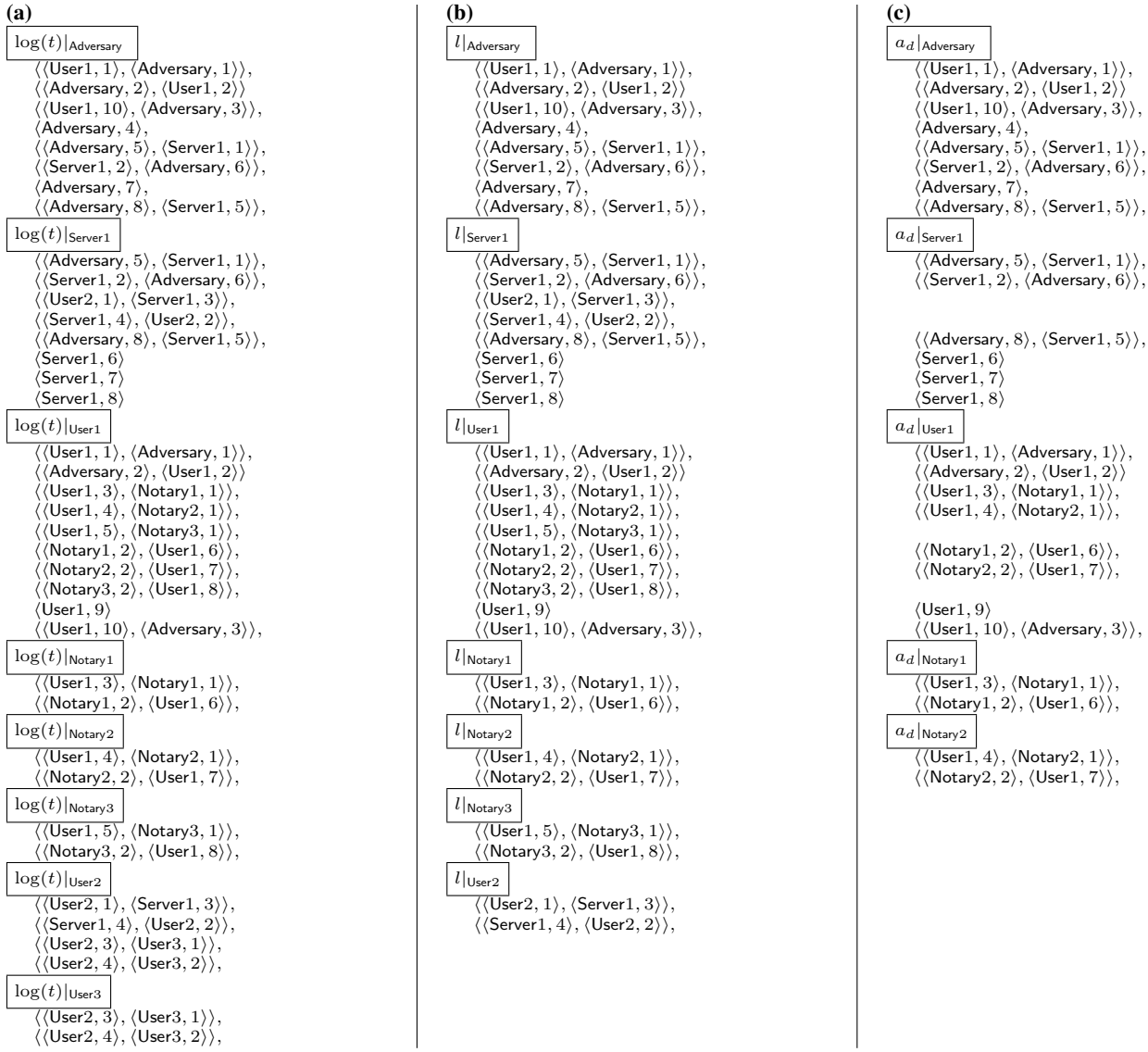

Figure 3. *Left to Right:* **(a):** $\log(t)|_i$ for $i \in I$. **(b):** Lamport cause l for Theorem 2. $l|_i = \emptyset$ for $i \in \{\mathsf{User3}\}$ as output by Definition 12. **(c):** Actual cause a_d for Theorem 2. $a_d|_i = \emptyset$ for $i \in \{\mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$. a_d is a projected *sublog* of Lamport cause l.

and $\{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary2}, \mathsf{Notary3}\}$ with the actual action causes given by a_d, a'_d and a''_d, respectively in Figure 3. These are symmetric, so we only explain why a_d satisfies Definition 14. (For this a_d, Definition 15 immediately forces $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$.) We show that (a) a_d satisfies sufficiency', and (b) No proper sublog of a_d satisfies sufficiency' (minimality'). Note that a_d is obtained from l by dropping Notary3, User2 and User3, and all their interactions with other threads.

We start with (a). Let a_d be such that $a_d|_i$ matches Figure 3(c) for every i. Fix any dummifying function f. We must show that any trace originating from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d as a projected sublog, is in φ_V. Additionally we must show that there is such a trace. There are two potential issues in mimicking the execution in a_d starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ — first, with the interaction between User1 and Notary3 and, second, with the interaction between Server1 and User2. For the first interaction, on line 5, $\mathcal{A}(\mathsf{User1})$ (Figure 2) synchronizes with Notary3 according to l, but the synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the `recv()` on line 8 in $\mathcal{A}(\mathsf{User1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, the majority check (assertion [B]) succeeds as in l, because two of the three notaries (Notary1 and Notary2) still attest the Adversary's key. A similar observation can be made about the interaction between Server1 and User2.

Next we prove that every trace starting from

$\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d (Figure 3) as a projected sublog, is in φ_V. Fix a trace t' with log l'. Assume l' contains a_d. We show $t' \in \varphi_V$ as follows:

1) Since the synchronization labels in l' are a superset of those in a_d, Server1 must execute line 8 of its program $\mathcal{A}(\mathsf{Server1})$ in t'. After this line, the access control matrix P contains $(acct, J)$ for some J.
2) When $\mathcal{A}(\mathsf{Server1})$ writes (x, J) to P at line 8, then J is the third component of a tuple obtained by decrypting a message received on line 5.
3) Since the synchronization projections on l' are a superset of a_d, and on a_d $\langle \mathsf{Server1}, 5\rangle$ synchronizes with $\langle \mathsf{Adversary}, 8\rangle$, J must be the third component of an encrypted message sent on line 8 of $\mathcal{A}(\mathsf{Adversary})$.
4) The third component of the message sent on line 8 by Adversary is exactly the term "Adversary". (This is easy to see, as the term "Adversary" is hardcoded on line 7.) Hence, $J = \mathsf{Adversary}$.
5) This immediately implies that $t' \in \varphi_V$ since $(acct, \mathsf{Adversary}) \in P$, but $\mathsf{Adversary} \neq \mathsf{User1}$.

Last, we prove (b) — that no proper subsequence of a_d satisfies sufficiency'. Note that a_d (Figure 3(c)) contains exactly those actions from l (Figure 3) on whose returned values the last statement of Server1's program (Figure 2) is data or control dependent. Consequently, all of a_d as shown is necessary to obtain the violation.

(The astute reader may note that in Figure 2, there is no dependency between line 1 of Server1's program and the `insert` statement in Server1. Hence, line 1 should not be in a_d. While this is accurate, the program in Figure 2 is a slight simplification of the real protocol, which is shown in the appendix. In the real protocol, line 1 returns a received nonce, whose value does influence whether or not execution proceeds to the `insert` statement.)

V. Towards Accountability

In this section, we discuss the use of our causal analysis techniques for providing explanations and assigning blame.

A. Using Causality for Explanations

Generating explanations involves enhancing the epistemic state of an agent by providing information about the cause of an outcome [32]. Automating this process is useful for several tasks such as planning in AI-related applications and has also been of interest in the philosophy community [32], [33]. Causation has also been applied for explaining counter examples and providing explanations for errors in model checking [34], [35], [36], [37] where the abstract nature of the explanation provides insight about the model.

In prior work, Halpern and Pearl have defined explanation in terms of causality [32]. A fact, say E, constitutes an explanation for a previously established fact F in a given context, if had E been true then it would have been a sufficient cause of the established fact F . Moreover, having this information advances the prior epistemic state of the agent seeking the explanation, i.e. there exists a world (or a setting of the variables in Halpern and Pearl's model) where F is not true but E is.

Our definition of cause (Section III) could be used to explain violations arising from execution of programs in a given initial configuration. Given a log l, an initial configuration $\mathcal{C}_0$, and a violation φ_V, our definition would pinpoint a sequence of program actions, a_d, as an actual cause of the violation on the log. a_d would also be an explanation for the violation on l if having this causal information advances the epistemic knowledge of the agent. Note that there could be traces arising from the initial configuration where the behavior is inconsistent with the log. Knowing that a_d is consistent with the behavior on the log and that it is a cause of the violation would advance the agent's knowledge and provide an explanation for the violation.

B. Using Causality for Blame Attribution

Actual causation is an integral part of the prominent theories of blame in social psychology and legal settings [38], [39], [31], [40]. Most of these theories provide a comprehensive framework for blame which integrates causality, intentionality and foreseeability [38], [39], [41]. These theories recognize blame and cause as interrelated yet distinct concepts. Prior to attributing blame to an actor, a causal relation must be established between the actor's actions and the outcome. However, not all actions which are determined as a cause are blameworthy and an agent can be blamed for an outcome even if their actions were not a direct cause (for instance if an agent was responsible for another agent's actions). In our work we focus on the first aspect where we develop a theory for actual causation and provide a building block to find blameworthy programs from this set.

We can use the causal set output by the definitions in Section III and further narrow down the set to find blameworthy programs. Note that in order to use our definition as a building block for blame assignment, we require information about a) which of the executed programs deviate from the protocol, and b) which of these deviations are harmless. Some harmless deviants might be output as part of the causal set because their interaction is critical for the violation to occur. Definition 17 below provides one approach to removing such non-blameworthy programs from the causal set. In addition we can filter the norms from the causal set.

For this purpose, we use the notion of protocol specified norms $\mathcal{N}$ introduced in Section IV. We impose an additional constraint on the norms, i.e., in the extreme counterfactual world where we execute norms only, there should be no possibility of violation. We call this condition *necessity*. Conceptually, necessity says that the reference standard (norms) we employ to assign blame is reasonable.

Definition 16 (Necessity condition for norms): Given $\langle I, \Sigma, \mathcal{N}, \varphi_V\rangle$, we say that $\mathcal{N}$ satisfies the necessity

condition w.r.t. φ_V if for any trace t' starting from the initial configuration $\langle I, \mathcal{N}, \Sigma \rangle$, it is the case that $t' \notin \varphi_V$.

We can use the norms $\mathcal{N}$ and the program cause X with its corresponding actual cause a_d from Phase 2 (Definitions 14, 15), in order to determine whether a program is a harmless deviant as follows. Definition 17 presents a sound (but not complete) approach for identifying harmless deviants.

Definition 17 (Harmless deviant): Let X be a program cause of violation V and a_d be the corresponding actual cause as determined by Definitions 14 and 15. We say that the program corresponding to index $i \in X$ is a harmless deviant w.r.t. trace t and violation φ_V if $\mathcal{A}(i)$ is deviant (i.e. $\mathcal{A}(i) \neq \mathcal{N}(i)$) and $a_d|_i$ is a prefix of $\mathcal{N}(i)$.

For instance in our case study (Section IV), Theorem 2 outputs X and a_d (Figure 3) as a cause. X includes Server1. Considering Server1's norm (Figure 1), $\mathcal{A}\{\mathsf{Server}\}$ will be considered a deviant, but according to Definition 17, Server1 will be classified as a *harmless deviant* because $a_d|_{\mathsf{Server1}}$ is a prefix of $\mathcal{N}(\mathsf{Server1})$. Note that in order to capture blame attribution accurately, we will need a richer model which incorporates intentionality, epistemic knowledge and foreseeability, beyond causality.

VI. Related Work

Currently, there are multiple proposals for providing accountability in decentralized multi-agent systems [23], [24], [11], [7], [10], [8], [42], [43], [44]. Although the intrinsic relationship between causation and accountability is often acknowledged, the foundational studies of accountability do not explicitly incorporate the notion of cause in their formal definition or treat it as a blackbox concept without explicitly defining it. Our thesis is that accountability is not a trace property since evidence from the log alone does not provide a justifiable basis to determine accountable parties. Actual causation is not a trace property; inferring actions which are actual causes of a violating trace requires analyzing counterfactual traces (see our sufficiency conditions). Accountability depends on actual causation and is, therefore, also not a trace property.

On the other hand, prior work on actual causation in analytical philosophy and AI has considered counterfactual based causation in detail [13], [14], [19], [20], [18], [15]. These ideas have been applied for fault diagnosis where system components are analyzed, but these frameworks do not adequately capture all the elements crucial to model a security setting. Executions in security settings involve interactions among concurrently running programs in the presence of adversaries, and little can be assumed about the scheduling of events. We discuss below those lines of work which are most closely related to ours.

Accountability: Küsters et al [11] define a protocol P with associated accountability constraints that are rules of the form: if a particular property holds over runs of the protocol instances then particular agents may be blamed. Further, they define a judge J who gives a verdict over a run r of an instance π of a protocol P, where the verdict blames agents. In their work, Küsters et al assume that the accountability constraints for each protocol are given and complete. They state that the judge J should be designed so that J's verdict is fair and complete w.r.t. these accountability constraints. They design a judge separately for every protocol with a specific accountability property. Küsters et al.'s definition of accountability has been successfully applied to substantial protocols such as voting, auctions, and contract signing. Our work complements this line of work in that we aim to provide a semantic basis for arriving at such accountability constraints, thereby providing a justification for the blame assignment suggested by those constraints. Our actual cause definition can be viewed as a generic judging procedure that is defined independent of the violation and the protocol. We believe that using our cause definition as the basis for accountability constraints would also ensure the minimality of verdicts given by the judges.

Backes et al [7] define accountability as the ability to show evidence when an agent deviates. The authors analyze a contract signing protocol using protocol composition logic. In particular, the authors consider the case when the trusted third-party acts dishonestly and prove that the party can be held accountable by looking at a violating trace. This work can be viewed as a special case of the subsequent work of Küsters et al. [11] where the property associated with the violating trace is an example of an accountability constraint.

Feigenbaum et al [23], [24] also propose a definition of accountability that focuses on linking a violation to punishment. They use Halpern and Pearl's definition [13], [14] of causality in order to define mediated punishment, where punishment is justified by the existence of a causal chain of events in addition to satisfaction of some utility conditions. The underlying ideas of our cause definition could be adapted to their framework to instantiate the causality notion that is currently used as a black box in their definition of mediated punishment. One key difference is that we focus on finding program actions that lead to the violation, which could explain why the violation happened while they focus on establishing a causal chain between violation and punishment events.

Causation for blame assignment: The work by Barth et al [42] provides a definition of accountability that uses the much coarser notion of Lamport causality, which is related to Phase 1 of our definition. However, we use minimality checks and filter out *progress enablers* in Phase 2 to obtain a finer determination of actual cause.

Gössler et al's work [43], [45] considers blame assignment for safety property violations where the violation of the global safety property implies that some components have violated their local specifications. They use a counterfactual notion of causality similar in spirit to ours to identify a subset of these faulty components as causes of the violation.

The most recent work in this line applies the framework to real-time systems specified using timed automata [46].

A key technical difference between this line of work and ours is the way in which the contingencies to be considered in counterfactual reasoning are constructed. We have a program-based approach to leverage reasoning methods based on invariants and program logics. Gössler et al assume that a dependency relation that captures information flow between component actions are given and construct their contingencies using the traces of faulty components observed on the log as a basis. A set of faulty components is the necessary cause of the violation if the violation would disappear once the traces of these faulty components are modified to match the components' local specifications. They determine the longest prefixes of faulty components that satisfy the specification and replace the faulty suffixes with a correct one. Doing such a replacement without taking into account its impact on the behavior of other components that interact with the faulty components would not be satisfactory. Indeed, Wang et al [44] describe a counterexample to Gössler et al's work [43] where all causes are not found because of not being able to completely capture the effect of one component's behavior on another's. The most recent definitions of Gössler et al [45], [46] address this issue by over approximating the parts of the log affected by the faulty components and replacing them with behavior that would have arisen had the faulty ones behaved correctly.

In constructing the contingencies to consider in counterfactual reasoning, we do not work with individual traces as Gössler et al. Instead, we work at the level of programs where "correcting" behavior is done by replacing program actions with those that do not have any effect on the violation other than enabling the programs to progress. The relevant contingencies follow directly from the execution of programs where such replacements have been done, without any need to develop additional machinery for reconstructing traces. Note also that we have a sufficiently fine-grained definition to pinpoint the minimal set of actions that make the component a part of the cause, where these actions may a part be of faulty or non-faulty programs. Moreover, we purposely separate cause determination and blame assignment because we believe that in the security setting, blame assignment is a problem that requires additional criteria to be considered such as the ability to make a choice, and intention. The work presented in this paper focuses on identifying cause as a *building block* for blame assignment.

VII. Conclusion

We have presented a first attempt at defining what it means for a sequence of program actions to be an actual cause of a violation of a security property. This question is motivated by security applications where agents can exercise their choice to either execute a prescribed program or deviate from it. While we demonstrate the value of this definition by analyzing a set of authentication failures, it would be interesting to explore applications to other protocols in which accountability concerns are central, in particular, protocols for electronic voting and secure multiparty computation in the semi-honest model. Another challenge in security settings is that deviant programs executed by malicious agents may not be available for analysis; rather there will be evidence about certain actions committed by such agents. A generalized treatment accounting for such partial observability would be technically interesting and useful for other practical applications. This work demonstrates the importance of program actions as causes as a useful *building block* for several such applications, in particular for providing explanations, assigning blame and providing accountability guarantees for security protocols.

References

[1] C. Kaufman, R. Perlman, and M. Speciner, *Network security: private communication in a public world*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[2] R. L. Rivest and W. D. Smith, "Three voting protocols: Threeballot, vav, and twin," in *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, ser. EVT'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 16–16.

[3] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe, "Practical secrecy-preserving, verifiably correct and trustworthy auctions," *Electron. Commer. Rec. Appl.*, vol. 7, no. 3, pp. 294–312, Nov. 2008.

[4] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 218–229.

[5] H. Nissenbaum, "Accountability in a computerized society," *Science and Engineering Ethics*, vol. 2, no. 1, pp. 25–42, 1996.

[6] B. Lampson, "Computer security in the real world," *Computer*, vol. 37, no. 6, pp. 37 – 46, june 2004.

[7] M. Backes, A. Datta, A. Derek, J. C. Mitchell, and M. Turuani, "Compositional analysis of contract-signing protocols," *Theor. Comput. Sci.*, vol. 367, no. 1-2, pp. 33–56, 2006.

[8] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," in *SOSP*, 2007, pp. 175–188.

[9] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman, "Information accountability," *Commun. ACM*, vol. 51, no. 6, pp. 82–87, Jun. 2008.

[10] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a theory of accountability and audit," in *ESORICS*, 2009, pp. 152–167.

[11] R. Küsters, T. Truderung, and A. Vogt, "Accountabiliy: Definition and Relationship to Verifiability," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM Press, 2010, pp. 526–535.

[12] C. Ellison, "Ceremony design and analysis." [Online]. Available: http://eprint.iacr.org/2007/399.pdf

[13] J. Y. Halpern and J. Pearl, "Causes and explanations: a structural-model approach: part i: causes," in *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, ser. UAI'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 194–202.

[14] ——, "Causes and explanations: A structural-model approach. part i: Causes," *British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 843–887, 2005.

[15] J. Pearl, *Causality: models, reasoning, and inference*. New York, NY, USA: Cambridge University Press, 2000.

[16] D. Hume, "An Enquiry Concerning Human Understanding," *Reprinted Open Court Press, LaSalle, IL, 1958*, 1748.

[17] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, no. 17, pp. 556–567, 1973.

[18] J. Collins, N. Hall, and L. A. Paul, *Causation and Counterfactuals*. MIT Press, 2004.

[19] J. L. Mackie, "Causes and Conditions," *American Philosophical Quarterly*, vol. 2, no. 4, pp. 245–264, 1965.

[20] R. Wright, "Causation in tort law," *California Law Review 73*, pp. 1735–1828, 1985.

[21] J. Gatins, "Flight," 2012. [Online]. Available: http://www.imdb.com/title/tt1907668/

[22] J. Y. Halpern, "Defaults and Normality in Causal Structures," *Artificial Intelligence*, vol. 30, pp. 198–208, 2008. [Online]. Available: http://arxiv.org/abs/0806.2140

[23] J. Feigenbaum, A. D. Jaggard, and R. N. Wright, "Towards a formal model of accountability," in *Proceedings of the 2011 workshop on New security paradigms workshop*, ser. NSPW '11. New York, NY, USA: ACM, 2011, pp. 45–56.

[24] J. Feigenbaum, J. A. Hendler, A. D. Jaggard, D. J. Weitzner, and R. N. Wright, "Accountability and deterrence in online life," in *Proceedings of the 3rd International Web Science Conference*, ser. WebSci '11. NY, USA: ACM, 2011, pp. 7:1–7:7.

[25] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: improving ssh-style host authentication with multi-path probing," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. CA, USA: USENIX Association, 2008.

[26] E. G. on Tort Law, *Principles of European Tort Law: Text and Commentary*. Springer, 2005. [Online]. Available: http://books.google.com/books?id=3Najct7xGuAC

[27] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[28] ——, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[29] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, 1985.

[30] W. Rafnsson, D. Hedin, and A. Sabelfeld, "Securing interactive programs," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, ser. CSF '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 293–307. [Online]. Available: http://dx.doi.org/10.1109/CSF.2012.15

[31] J. Feinberg, "Ethical issues in the use of computers," D. G. Johnson and J. W. Snapper, Eds. Belmont, CA, USA: Wadsworth Publ. Co., 1985, ch. Sua Culpa, pp. 102–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=2569.2675

[32] J. Y. Halpern and J. Pearl, "Causes and explanations: A structural-model approach. part ii: Explanations," *The British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 889–911, 2005.

[33] J. Woodward, *Making things happen: A theory of causal explanation*. Oxford University Press, 2003.

[34] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 94–108. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_11

[35] H. Chockler, J. Y. Halpern, and O. Kupferman, "What causes a system to satisfy a specification?" *ACM Trans. Comput. Logic*, vol. 9, pp. 20:1–20:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1352582.1352588

[36] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 229–247, 2006.

[37] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03. New York, NY, USA: ACM, 2003, pp. 97–105. [Online]. Available: http://doi.acm.org/10.1145/604131.604140

[38] K. Shaver, *The attribution of blame: Causality, responsibility, and blameworthiness*. Springer Science & Business Media, 2012.

[39] M. D. Alicke, "Culpable control and the psychology of blame." *Psychological bulletin*, vol. 126, no. 4, p. 556, 2000.

[40] L. Kenner, "On blaming," *Mind*, pp. 238–249, 1967.

[41] D. A. Lagnado and S. Channon, "Judgments of cause and blame: The effects of intentionality and foreseeability," *Cognition*, vol. 108, no. 3, pp. 754–770, 2008.

[42] A. Barth, J. C. Mitchell, A. Datta, and S. Sundaram, "Privacy and utility in business processes," in *CSF*, 2007, pp. 279–294.

[43] G. Gössler, D. Le Métayer, and J.-B. Raclet, "Causality analysis in contract violation," in *Proceedings of the First International Conference on Runtime Verification*, ser. RV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 270–284.

[44] S. Wang, A. Ayoub, R. Ivanov, O. Sokolsky, and I. Lee, "Contract-based blame assignment by trace analysis," in *Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems*. NY, USA: ACM, 2013.

[45] G. Gössler and D. L. Métayer, "A general trace-based framework of logical causality," in *Formal Aspects of Component Software - 10th International Symposium, FACS 2013*, 2013, pp. 157–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07602-7_11

[46] G. Gössler and L. Aştefănoaei, "Blaming in component-based real-time systems," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14. ACM, 2014, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/2656045.2656048

APPENDIX

A. Operational Semantics

Selected rules of the operational semantics of the programming language L are shown below.

$\boxed{T \hookrightarrow T'}$

$$\frac{\texttt{eval}\ t\ t'}{I\,,t\,,\sigma \overset{\epsilon}{\hookrightarrow} I\,,t'\,,\sigma'}\text{red-end} \qquad \frac{\sigma;\zeta(t) \mapsto \sigma';t' \qquad \texttt{eval}\ t'\ t''}{I\,,((b:x=\zeta(t));e)\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{t''/x\}\,,\sigma'}\text{red-act}$$

$$\frac{\texttt{eval}\ t\ \texttt{true}}{I\,,(\texttt{assert}(t);e)\,,\sigma \overset{\epsilon}{\hookrightarrow} I\,,e\,,\sigma}\text{red-assert}$$

$\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

Internal reduction

$$\frac{T_i \overset{r}{\hookrightarrow} T_i'}{\ldots,T_i,\ldots \overset{r}{\longrightarrow} \ldots,T_i',\ldots}\text{red-config}$$

Communication action

$$\frac{\texttt{eval}\ t\ t'}{\ldots,\langle I_s\,,((b_s:x=\texttt{send}(t));e_s)\,,\sigma_s\rangle,\langle I_r\,,((b_r:y=\texttt{recv}());e_r)\,,\sigma_r\rangle,\ldots \overset{\langle\langle I_s,b_s\rangle,\langle I_r,b_r\rangle\rangle}{\longrightarrow} \ldots,\langle I_s\,,e_s[0/x]\,,\sigma_s\rangle,\langle I_r\,,e_r[t'/y]\,,\sigma_r\rangle,\ldots}\text{red-comm}$$

B. Case study: Compromised notaries attack

We model an instance of our running example based on passwords in order to demonstrate our actual cause definition. As explained in Section II, we consider a protocol session where Server1, User1, User2, User3 and multiple notaries interact over an adversarial network to establish access over a password-protected account. In parallel for this scenario, we assume the log also contains interactions of a second server (Server2), one notary (Notary4, not contacted by User1, User2 or User3) and another user (User4) who follow their norms for account access. These threads do not interact with threads {User1, Server1, Notary1, Notary2, Notary3, Adversary, User2, User3}. The protocol has been described in detail below.

1) Protocol Description: We consider our example protocol with eleven threads named {Server1, User1, User2, User3, Adversary, Notary1, Notary2, Notary3, Notary4, Server2, User4}. The *norms* for all these threads, except Adversary are shown in Figure 4. The actual violation is caused because some of the executing programs are different from the norms. These actual programs, called $\mathcal{A}$ as in Section III, are shown later. The norms are shown here to help the reader understand what the ideal protocol is.

In this case study, we have two servers (Server1, Server2) running the protocol with two different users (User1, User4) and each server allocates account access separately. The norms in Figure 4 assume that User1's and User4's accounts (called $acct_1$ and $acct_2$ in Server1's and Server2's norm respectively) have been created already. User1's password, pwd_1 is associated with User1's user id $uid1$. Similarly User4's password pwd_2 is associated with its user id $uid2$. This association (in hashed form) is stored in Server1's local state at pointer mem_1 (and at mem_2 for Server2). The norm for Server1 is to wait for a request from an entity, respond with its public key, then wait for a password encrypted with that public key and grant access to the requester if the password matches the previously stored value in Server1's memory at mem_1. To grant access, Server1 adds an entry into a private access matrix, called P_1. (A separate server thread, not shown here, allows User1 to access its resource if this entry exists in P_1.)

The norm for User1 is to send an access request to Server1, wait for the server's public key, verify that key with three notaries and then send its password pwd_1 to Server1, encrypted under Server1's public key. On receiving Server1's public key, User1 initiates a protocol with the three notaries and accepts or rejects the key based on the response of a majority of the notaries.

The norm for User4 is the same as that for User1 except that it interacts with Server2. Note that User4 only verifies the public key with one notary, Notary4. The norm for Server2 is the same as that for Server1 except that it interacts with User4.

In parallel, the norm for User2 is to generate and send a nonce to User3. The norm for User3 is to receive a message from User2, generate a nonce and send it to User2.

Each notary has a private database of *(public_key, principal)* tuples. The norms here assume that this database has already been created correctly. When User1 or User4 send a request with a public key, the notary responds with the principal's identifier after retrieving the tuple corresponding to the key in its database. (Note that, in this simple example, we identify threads with principals, so the notaries just store an association between public keys and their threads.)

2) Preliminaries:

Notation: The programs in this example use several primitive functions ζ. $\texttt{Enc}(k, m)$ and $\texttt{Dec}(k', m)$ denote encryption and decryption of message m with key k and k' respectively. $\texttt{Hash}(m)$ generates the hash of term m. $\texttt{Sig}(k, m)$ denotes message m signed with the key k, paired with m in the clear. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For readability, we include the intended recipient i and expected sender j of a message as the first argument of $\texttt{send}(i, m)$ and $\texttt{recv}(j)$ expressions. As explained earlier, i and j are ignored during execution and a network adversary, if present, may capture or inject any messages. $\texttt{Send}(i, j, m)$ @ u holds if thread i sends message m to thread j at time u and $\texttt{Recv}(i, j, m)$ @ u hold if thread i receives message m from thread j at time u. $P_1(u)$ and $P_2(u)$ denotes the tuples in the permission matrices at time u. Initially P_1 and P_2 do not contain any access permissions.

Assumptions: (A1)

$$\texttt{HonestThread}(\mathsf{Server1}, \mathcal{A}(\mathsf{Server1}))$$

We are interested in security guarantees about users who create accounts by interacting with the server and who do not share the generated password or user-id with any other principal except for sending it according to the roles specified in the program given below.

(A2)

$$\texttt{HonestThread}(\mathsf{User1}, \mathcal{A}(\mathsf{User1}))$$

(A3)

$$\texttt{HonestThread}(\mathsf{Adversary}, \mathcal{A}(\mathsf{Adversary}))$$

(A4)

$$\texttt{HonestThread}(\mathsf{Notary1}, \mathcal{A}(\mathsf{Notary1}))$$

(A5)

$$\texttt{HonestThread}(\mathsf{Notary2}, \mathcal{A}(\mathsf{Notary2}))$$

(A6)

$$\texttt{HonestThread}(\mathsf{Notary3}, \mathcal{A}(\mathsf{Notary3}))$$

(A7)

$$\texttt{HonestThread}(\mathsf{Notary4}, \mathcal{A}(\mathsf{Notary4}))$$

(A8)

$$\texttt{HonestThread}(\mathsf{Server2}, \mathcal{A}(\mathsf{Server2}))$$

(A9)

$$\texttt{HonestThread}(\mathsf{User4}, \mathcal{A}(\mathsf{User4}))$$

(A10)

$$\texttt{HonestThread}(\mathsf{User2}, \mathcal{A}(\mathsf{User2}))$$

(A11)

$$\texttt{HonestThread}(\mathsf{User3}, \mathcal{A}(\mathsf{User3}))$$

A principal following the protocol never shares its keys with any other entity. We also assume that the encryption scheme in semantically secure and non-malleable. Since we identify threads with principals therefore each of the threads are owned by principals with the same identifier, for instance Server1 owns the thread that executes the program A(Server1).

(Start1)

$$Start(i) \ @ \ -\infty$$

where i refers to all the threads in the set described above.

Norm $\mathcal{N}$(Server1):

$1 : (uid1, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, (pub\_key\_\mathsf{Server1}, n2, n1))$; //sign and send public key
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid1, pwd1$ from j, alongwith its thread id J
$5 : (n3, uid1, pwd1, J) = \texttt{Dec}(pvt\_key\_\mathsf{Server1}, s1)$;
$6 : t = \texttt{Hash}(uid1, pwd1)$;
$\texttt{assert}(mem_1 = t)$ //compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_1, (acct_1, J))$;

Norm $\mathcal{N}$(User1):

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\mathsf{Server1}, (uid1, n1))$; //access request
$3 : (pub\_key1, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3, n4, n5 = \texttt{new}$;
$5 : \texttt{send}(\mathsf{Notary1}, pub\_key1, n3)$;
$6 : \texttt{send}(\mathsf{Notary2}, pub\_key1, n4)$;
$7 : \texttt{send}(\mathsf{Notary3}, pub\_key1, n5)$;
$8 : \texttt{Sig}(pvt\_key\_\mathsf{Notary1}, (pub\_key1, l1, n3)) = \texttt{recv}(\mathsf{Notary1})$; //notary1 responds
$9 : \texttt{Sig}(pvt\_key\_\mathsf{Notary2}, (pub\_key1, l2, n4)) = \texttt{recv}(\mathsf{Notary2})$; //notary2 responds
$10 : \texttt{Sig}(pvt\_key\_\mathsf{Notary3}, (pub\_key1, l3, n5)) = \texttt{recv}(\mathsf{Notary3})$; //notary3 responds
$\texttt{assert}$(At least two of {l1,l2,l3} equal Server1)
$11 : t = \texttt{Enc}(pub\_key1, n2, (uid1, pwd1, \mathsf{User1}))$;
$12 : \texttt{send}(\mathsf{Server1}, t)$; //send t to Server1;

Norms $\mathcal{N}$(Notary1), $\mathcal{N}$(Notary2), $\mathcal{N}$(Notary3), $\mathcal{N}$(Notary4):

// o denotes Notary1, Notary2, Notary3 or Notary4
$1 : (pub\_key, n1) = \texttt{recv}(j)$;
$2 : pr = \texttt{KeyOwner}(pub\_key)$; //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr, n1)))$; //signed certificate;

Norm $\mathcal{N}$(Server2):

$1 : (uid2, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, (pub\_key\_\mathsf{Server2}, n2, n1))$;
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid2, pwd2$ from j, alongwith its thread id J
$5 : (n2, uid2, pwd2, J) = \texttt{Dec}(pvt\_key\_\mathsf{Server2}, s1)$;
$6 : t = \texttt{Hash}(uid2, pwd2)$;
$\texttt{assert}(mem_2 = t)$ //compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_2, (acct_2, J))$;

Norm $\mathcal{N}$(User4):

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\mathsf{Server2}, (uid2, n1))$; //access request
$3 : pub\_key, n2, n1 = \texttt{recv}(j)$; //key from j
$4 : n3 = \texttt{new}$;
$5 : \texttt{send}(\mathsf{Notary4}, pub\_key, n3)$;
$6 : \texttt{Sig}(pvt\_key\_\mathsf{Notary4}, (pub\_key, l1, n3)) = \texttt{recv}(\mathsf{Notary4})$; //notary4 responds
$\texttt{assert}(\{l1\}$ equals Server2)
$7 : t = \texttt{Enc}(pub\_key, n2, (uid2, pwd2, \mathsf{User4}))$;
$8 : \texttt{send}(\mathsf{Server2}, t)$; //send t to Server2;

Norm $\mathcal{N}$(User2):

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\mathsf{User3}, (n1))$;
$3 : \texttt{Sig}(pvt\_key\_j, (n2, n1)) = \texttt{recv}(\mathsf{User3}); 4 :$

Norm $\mathcal{N}$(User3):

$1 : n1 = \texttt{recv}(\mathsf{User2})$;
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\mathsf{User3}, \texttt{Sig}(pvt\_key\_\mathsf{User3}, (n2, n1)))$;

Figure 4. Norms for Server1, User1, Server2, User4, User2, User3 and the notaries. Adversary's norm is the trivial empty program.

Security property: The security property of interest to us is that if at time u, a thread k is given access to account a, then k owns a. Specifically, in this example, we are interested in the $a = acct_1$ and $k = \mathsf{User1}$. This can be formalized by the following logical formula, $\neg\varphi_V$:

$$\forall u, k.\ (acct_1, k) \in P_1(u) \supset (k = \mathsf{User1}) \tag{2}$$

Here, $P_1(u)$ is the state of the access control matrix P_1 for Server1 at time u.

The actuals for all threads are shown in Figure 5 and 6.

3) Attack: As an illustration, we model the "Compromised Notaries" violation of Section II. The programs executed by all threads are given in Figures 5 and 6. User1 sends an access request to Server1 which is intercepted by Adversary who sends its own key to User1 (pretending to be Server1). User1 checks with the three notaries who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to the User1's account. Note that the actual programs of the three notaries attest that the public key given to them belongs to Server1. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 interacts with User3, as in their norms. User4, Server2 and Notary4 execute their actuals in order to access the account $acct_2$ as well.

Figure 7 shows the expressions executed by each thread on the property-violating trace. For instance, the label $\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle$ indicates that both User1 and Adversary executed the expressions with the line number 1 in their actual programs, which resulted in a synchronous communication between them, while the label $\langle \mathsf{Adversary}, 4\rangle$ indicates the local execution of the expression at line 4 of Adversary's program. The initial configuration has the programs: $\{\mathcal{A}(\mathsf{User1}), \mathcal{A}(\mathsf{Server1}), \mathcal{A}(\mathsf{Adversary}), \mathcal{A}(\mathsf{Notary1}), \mathcal{A}(\mathsf{Notary2}), \mathcal{A}(\mathsf{Notary3}), \mathcal{A}(\mathsf{User2}), \mathcal{A}(\mathsf{User3}), \mathcal{A}(\mathsf{User4}), \mathcal{A}(\mathsf{Server2}), \mathcal{A}(\mathsf{Notary4})\}$. For this attack scenario, the concrete trace t we consider is such that $\log(t)$ is any *arbitrary interleaving* of the actions for $X_1 = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$ and $X_2 = \{\mathsf{Server2}, \mathsf{User4}, \mathsf{Notary4}\}$ shown in Figure 7(a) and Figure 8. Any such interleaved log is denoted $\log(t)$ in the sequel. At the end of this log, $(acct_1, \mathsf{Adversary})$ occurs in the access control matrix P_1, but Adversary does not own $acct_1$. Hence, this log corresponds to a violation of our security property.

Note that, if any two of the three notaries had attested the Adversary's key to belong to Server1, the violation would have still happened. Consequently, we may expect three independent program causes in this example: {Adversary, User1, Server1, Notary1, Notary2} with the action causes a_d as shown in Figure 7(c), {Adversary, User1, Server1, Notary1, Notary3} with the actions a'_d, and {Adversary, User1, Server1, Notary2, Notary3} with the actions a''_d where a'_d and a''_d can be obtained from a_d (Figure 7(c)) by considering actions for {Notary1, Notary3} and {Notary2, Notary3} respectively, instead of actions for {Notary1, Notary2}. The following theorem states that our definitions determine exactly these three independent causes.

Theorem 3: Let $I = \{\mathsf{User1}, \mathsf{Server1}, \mathsf{Adversary}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{Notary4}, \mathsf{Server2}, \mathsf{User4}, \mathsf{User2}, \mathsf{User3}\}$, and Σ and $\mathcal{A}$ be as described above. Let t be a trace from $\langle I, \mathcal{A}, \Sigma\rangle$ such that $log(t)|_i$ for each $i \in I$ matches the corresponding log projection from Figures 7(a) and 8. Then, Definition 15 determines three possible values for the program cause X of violation $t \in \varphi_V$: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3} where the corresponding actual causes are a_d, a'_d and a''_d respectively.

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that our Phase 1 and Phase 2 definitions (Definitions 12, 14, 15) yield exactly the three values for X mentioned in the theorem.

Lamport cause (Phase 1): We show that any l whose projections match those shown in Figure 7(b) satisfies sufficiency and minimality. From Figure 7(b), such an l has no actions for User3, User4, Notary4, Server2 and only those actions of User2 that are involved in synchronization with Server1. For all other threads, the log contains every action from t. The intuitive explanation for this l is straightforward: Since l must be a (projected) *prefix* of the trace, and the violation only happens because of `insert` in the last statement of Server1's program, every action of every program before that statement in Lamport's happens-before relation must be in l. This is exactly the l described in Figure 7(b).

Formally, following the statement of sufficiency, let T be the set of traces starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$ (Figure 5) whose logs contain l as a projected prefix. Pick any $t' \in T$. We need to show $t' \in \varphi_V$. However, note that any t' containing all actions in l must also add $(acct_1, \mathsf{Adversary})$ to P_1, but Adversary $\neq$ User1. Hence, $t' \in \varphi_V$. Further, l is minimal as described in the previous paragraph.

Actual cause (Phase 2): Phase 2 (Definitions 14, 15) determines three independent program causes for X: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3} with the actual action causes given by a_d, a'_d and a''_d, respectively in Figure 7(c). These are symmetric, so we only explain why a_d satisfies Definition 14. (For this a_d, Definition 15 immediately forces $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$.) We show that (a) a_d satisfies sufficiency', and (b) No proper *sublog*

Actual $\mathcal{A}$(Adversary)

$1 : (uid1, n1) = \texttt{recv}(j)$; //intercept req from User1
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\text{User1}, (pub\_key\_\text{Adversary1}, n2, n1))$; //send key to User1
$4 : s = \texttt{recv}(\text{User1})$; //pwd from User
$5 : n2, uid1, pwd1, \text{User1} = \texttt{Dec}(pvt\_key\_\text{Adversary}, s)$; //decrypt pwd;
$6 : n3 = \texttt{new}$;
$7 : \texttt{send}(\text{Server1}, (uid1, n3))$; //access request to Server
$8 : pub\_key, n4, n3 = \texttt{recv}(\text{Server1})$;
$9 : t = \texttt{Enc}(pub\_key, (n4, uid1, pwd1, \text{Adversary}))$; //encrypt pwd
$10 : \texttt{send}(\text{Server1}, t)$; //pwd to Server1

Actuals $\mathcal{A}$(Notary1), $\mathcal{A}$(Notary2), $\mathcal{A}$(Notary3)**:**

// o denotes Notary1, Notary2 or Notary3
$1 : (pub\_key\_\text{Adversary}, n1) = \texttt{recv}(j)$;
$2 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key\_\text{Adversary}, \text{Server1}, n1))$; //signed certificate to j;

Actual $\mathcal{A}$(Server1)**:**

$1 : (uid1, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, (pub\_key\_\text{Server1}, n2, n1))$;
$4 : n4 = \texttt{recv}(j)$; //receive nonce from thread User2
$5 : n5 = \texttt{new}$;
$6 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_\text{Server1}, (n5, n4)))$;
$7 : s1 = \texttt{recv}(j)$; //encrypted $uid1, pwd1$ from j, alongwith its thread id J
$8 : (n3, uid1, pwd1, J) = \texttt{Dec}(pvt\_key\_\text{Server1}, s1)$;
$9 : t = \texttt{Hash}(uid1, pwd1)$;
$\texttt{assert}(mem_1 = t)$**[A]** //compare hash with stored hash value for same uid
$10 : \texttt{insert}(P_1, (acct_1, J))$;

Actual $\mathcal{A}$(User1)**:**

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server1}, (uid1, n1))$; //access request
$3 : (pub\_key, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3, n4, n5 = \texttt{new}$;
$5 : \texttt{send}(\text{Notary1}, pub\_key, n3)$;
$6 : \texttt{send}(\text{Notary2}, pub\_key, n4)$;
$7 : \texttt{send}(\text{Notary3}, pub\_key, n5)$;
$8 : \texttt{Sig}(pvt\_key\_\text{Notary1}, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$9 : \texttt{Sig}(pvt\_key\_\text{Notary2}, (pub\_key, l2, n4)) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$10 : \texttt{Sig}(pvt\_key\_\text{Notary3}, (pub\_key, l3, n5)) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$\texttt{assert}(\text{At least two of}\{l1, l2, l3\}\text{equal Server1})$; **[B]** //
$11 : t = \texttt{Enc}(pub\_key, n2, (uid1, pwd1, \text{User1}))$;
$12 : \texttt{send}(\text{Server1}, t)$; //send t to Server1;

Actual $\mathcal{A}$(User2)**:**

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server1}, (n1))$;
$3 : \texttt{Sig}(pvt\_key, (n2, n1)) = \texttt{recv}(\text{Server1})$;
$4 : \texttt{send}(\text{User3}, (n2))$;
$5 : \texttt{Sig}(pub\_key, n3, n2) = \texttt{recv}(\text{User3})$;

Actual $\mathcal{A}$(User3)**:**

$1 : n1 = \texttt{recv}(\text{User2})$;
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\text{User3}, \texttt{Sig}(pvt\_key\_\text{User3}, n2, n1))$;

Figure 5. Actuals for Adversary, Notary1, Notary2, Notary3, Server1, User1, User2, User3

Actual $\mathcal{A}(\mathsf{Server2})$**:**

$1 : (uid2, n1) = \texttt{recv}(j);$ //access req from thread j
$2 : n2 = \texttt{new};$
$3 : \texttt{send}(j, (pub\_key\_\mathsf{Server2}, n2, n1));$
$4 : s1 = \texttt{recv}(j);$ //encrypted $uid2, pwd2$ from j, alongwith its thread id J
$5 : (n2, uid2, pwd2, J) = \texttt{Dec}(pvt\_key\_\mathsf{Server2}, s1);$
$6 : t = \texttt{Hash}(uid2, pwd2);$
$\texttt{assert}(mem_2 = t)$ //(C)compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_2, (acct_2, J));$

Actual $\mathcal{A}(\mathsf{User4})$**:**

$1 : n1 = \texttt{new};$
$2 : \texttt{send}(\mathsf{Server2}, (uid2, n1));$ //access request
$3 : \texttt{Sig}(pub\_key, n2, n1) = \texttt{recv}(j);$ //key from j
$4 : n3 = \texttt{new};$
$5 : \texttt{send}(\mathsf{Notary4}, pub\_key, n3);$
$6 : \texttt{Sig}(pvt\_key\_\mathsf{Notary4}, (pub\_key, l1, n3)) = \texttt{recv}(\mathsf{Notary4});$ //notary4 responds
$\texttt{assert}(\{\mathrm{l1}\}\ \text{equals}\ \mathsf{Server2})(D)$
$7 : t = \texttt{Enc}(pub\_key, n2, (uid2, pwd2, \mathsf{User4}));$
$8 : \texttt{send}(\mathsf{Server2}, t);$ //send t to Server2;

Actual $\mathcal{A}(\mathsf{Notary4})$**:**

// o denotes Notary1, Notary2, Notary3 or Notary4
$1 : (pub\_key, n1) = \texttt{recv}(j);$
$2 : pr = \texttt{KeyOwner}(pub\_key);$ //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr, n1)));$ //signed certificate;

Figure 6. Actuals for Server2, User4, Notary4

of a_d satisfies sufficiency' (minimality'). Note that a_d is obtained from l by dropping Notary3, User2 and User3, and all their interactions with other threads.

We start with (a). Let a_d be such that $a_d|_i$ matches Figure 7(c) for every i. Fix any dummifying function f. We must show that any trace originating from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d as a projected sublog, is in φ_V. Additionally we must show that there is such a trace. There are two potential issues in mimicking the execution in a_d starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ — first, with the interaction between User1 and Notary3 and, second, with the interaction between Server1 and User2. For the first interaction, on line 7, $\mathcal{A}(\mathsf{User1})$ (Figure 5) synchronizes with Notary3 according to l, but the synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the $\texttt{recv}()$ on line 10 in $\mathcal{A}(\mathsf{User1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, the majority check (assertion [B]) succeeds as in l, because two of the three notaries (Notary1 and Notary2) still attest the Adversary's key.

A similar observation can be made about the interaction between Server1 and User2. Line 4, $\mathcal{A}(\mathsf{Server1})$ (from Figure 7(b)) synchronizes with User2 according to l, but this synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the $\texttt{recv}()$ on line 4 in $\mathcal{A}(\mathsf{Server1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, Server1 still adds permission for the Adversary.

Next we prove that every trace starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d (Figure 7(c)) as a projected *sublog*, is in φ_V. Fix a trace t' with log l'. Assume l' coincides with a_d. We show $t' \in \varphi_V$ as follows:

1) Since the synchronization labels in l' are a superset of those in a_d, Server1 must execute line 10 of its program $\mathcal{A}(\mathsf{Server1})$ in t'. After this line, the access control matrix P_1 contains $(acct_1, J)$ for some J.
2) When $\mathcal{A}(\mathsf{Server1})$ writes (x, J) to P_1 at line 10, then J is the third component of a tuple obtained by decrypting a message received on line 7.
3) Since the synchronization projections on l' are a superset of a_d, and on a_d $\langle \mathsf{Server1}, 7\rangle$ synchronizes with $\langle \mathsf{Adversary}, 10\rangle$, J must be the third component of an encrypted message sent on line 10 of $\mathcal{A}(\mathsf{Adversary})$.
4) The third component of the message sent on line 10 by Adversary is exactly the term "Adversary". (This is easy to see, as the term "Adversary" is hardcoded on line 9.) Hence, $J = \mathsf{Adversary}$.
5) This immediately implies that $t' \in \varphi_V$ since $(acct_1, \mathsf{Adversary}) \in P_1$, but $\mathsf{Adversary} \neq \mathsf{User1}$.

Last, we prove (b) — that no proper subsequence of a_d satisfies sufficiency'. Note that a_d (Figure 7(c)) contains exactly those actions from l (Figure 7) on whose returned values the last statement of Server1's program (Figure 5) is data or control dependent. Consequently, all of a_d as shown is necessary to obtain the violation.

(a)

$\log(t)|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$\log(t)|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle,$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$
$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$\log(t)|_{\mathsf{Server1}}$**:**
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle \mathsf{Server1}, 5\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$\log(t)|_{\mathsf{Notary1}}$**:**
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$\log(t)|_{\mathsf{Notary2}}$**:**
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$\log(t)|_{\mathsf{Notary3}}$**:**
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$

$\log(t)|_{\mathsf{User2}}$**:**
$\langle \mathsf{User2}, 1\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle\langle \mathsf{User3}, 3\rangle, \langle \mathsf{User2}, 5\rangle\rangle,$

$\log(t)|_{\mathsf{User3}}$**:**
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle \mathsf{User3}, 2\rangle,$
$\langle\langle \mathsf{User3}, 3\rangle, \langle \mathsf{User2}, 5\rangle\rangle,$

(b)

$l|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$l|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$
$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$l|_{\mathsf{Server1}}$**:**
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle \mathsf{Server1}, 5\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$l|_{\mathsf{Notary1}}$**:**
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$l|_{\mathsf{Notary2}}$**:**
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$l|_{\mathsf{Notary3}}$**:**
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$

$l|_{\mathsf{User2}}$**:**
$\langle \mathsf{User2}, 1\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$

(c)

$a_d|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$a_d|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$

$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$a_d|_{\mathsf{Server1}}$**:**
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$

$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$a_d|_{\mathsf{Notary1}}$**:**
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$a_d|_{\mathsf{Notary2}}$**:**
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

Figure 7. *Left to Right:* **(a):** $\log(t)|_i$ for $i \in \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$. **(b):** Lamport cause l for Theorem 3. $l|_i = \emptyset$ for $i \in \{\mathsf{Notary4}, \mathsf{Server2}, \mathsf{User4}, \mathsf{User3}\}$ as output by Definition 12. **(c):** Actual cause a_d for Theorem 3. $a_d|_i = \emptyset$ for $i \in \{\mathsf{Notary3}, \mathsf{Notary4}, \mathsf{Server2}, \mathsf{User4}, \mathsf{User2}, \mathsf{User3}\}$. a_d is a projected *sublog* of Lamport cause l.

$\log(t)|_{\mathsf{Server2}}$:
$\langle\langle \mathsf{User}4, 2\rangle, \langle \mathsf{Server}2, 1\rangle\rangle$,
$\langle \mathsf{Server}2, 2\rangle$,
$\langle\langle \mathsf{Server}2, 3\rangle, \langle \mathsf{User}4, 3\rangle\rangle$,
$\langle\langle \mathsf{User}4, 8\rangle, \langle \mathsf{Server}2, 4\rangle\rangle$,
$\langle \mathsf{Server}2, 5\rangle$,
$\langle \mathsf{Server}2, 6\rangle$,
$\langle \mathsf{Server}2, 7\rangle$,

$\log(t)|_{\mathsf{User1}}$
$\langle \mathsf{User}4, 1\rangle$,
$\langle\langle \mathsf{User}4, 2\rangle, \langle \mathsf{Server}2, 1\rangle\rangle$,
$\langle\langle \mathsf{Server}2, 3\rangle, \langle \mathsf{User}4, 3\rangle\rangle$,
$\langle \mathsf{User}4, 4\rangle$,
$\langle\langle \mathsf{User}4, 5\rangle, \langle \mathsf{Notary}4, 1\rangle\rangle$,
$\langle\langle \mathsf{Notary}4, 3\rangle, \langle \mathsf{User}4, 6\rangle\rangle$,
$\langle \mathsf{User}4, 7\rangle$,
$\langle\langle \mathsf{User}4, 8\rangle, \langle \mathsf{Server}2, 4\rangle\rangle$,

$\log(t)|_{\mathsf{Notary4}}$:
$\langle\langle \mathsf{User}4, 5\rangle, \langle \mathsf{Notary}4, 1\rangle\rangle$,
$\langle \mathsf{Notary}4, 2\rangle$,
$\langle\langle \mathsf{Notary}4, 3\rangle, \langle \mathsf{User}4, 6\rangle\rangle$,

Figure 8. $\log(t)|_i$ where $i \in \{\mathsf{User}4, \mathsf{Server}2, \mathsf{Notary}4\}$

In particular, observe that if labels for Server1 ($a_d|_{\mathsf{Server1}}$) are not a part of a'_d, then Server1's labels are not in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ and, hence, on any counterfactual trace Server1 cannot write to P_1, thus precluding a violation. Therefore, the sequence of labels in $a_d|_{\mathsf{Server1}}$ are required in the actual cause.

By sufficiency', for any f, the log of trace t' of $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ must contain a_d as a projected *sublog*. This means that in t', the assertion [A] of $\mathcal{A}(\mathsf{Server}1)$ must succeed and, hence, on line 7, the correct password pwd_1 must be received by Server1, independent of f. This immediately implies that Adversary's action of sending that password must be in a_d, else some dummified executions will have the wrong password sent to Server1 and the assertion [A] will fail.

Extending this logic further, we now observe that because Adversary forwards a password received from User1 (line 4 of $\mathcal{A}(\mathsf{Adversary})$) to Server1, the send action of User1 will be in a_d (otherwise, some dummifications of line 4 of $\mathcal{A}(\mathsf{Adversary})$ will result in the wrong password being sent to Server1, a contradiction). Since User1's action is in a_d and l' must contain a_d as a *sublog*, the majority check of $\mathcal{A}(\mathsf{User}1)$ must also succeed. This means that at least two of $\{\mathsf{Notary}1, \mathsf{Notary}2, \mathsf{Notary}3\}$ must send the confirmation to User1, else the dummification of lines 8 – 10 of $\mathcal{N}(\mathsf{User}1)$ will cause the assertion [B] to fail for some f. Since we are looking for a minimal *sublog* therefore we only consider the send actions from two threads i.e. $\{\mathsf{Notary}1, \mathsf{Notary}2\}$. At this point we have established that each of the labels as shown in Figure 7(c) are required in a_d. Hence, $a'_d = a_d$.