Programs as Actual Causes
A Building Block for Accountability

Anupam Datta\textsuperscript{1}, Deepak Garg\textsuperscript{2}, Dilsun Kaynar\textsuperscript{1}, Divya Sharma\textsuperscript{1}, and Arunesh Sinha\textsuperscript{3}

\textsuperscript{1} Carnegie Mellon University
\{danupam, dilsunk, divyasharma\}@cmu.edu
\textsuperscript{2} Max Planck Institute for Software Systems
dg@mpi-sws.org
\textsuperscript{3} University of Southern California
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 programs as actual causes. Specifically, we define what it means for a set of programs to be an actual cause of a violation when they are run concurrently with a set of other programs. Our definitions are inspired by prior work on counterfactual-based actual causation \cite{14, 16} that defines what it means for an event $c$ to be an actual cause of an event $e$. Considering programs instead of events as actual causes is appropriate in security settings because individual agents can exercise their choice to either execute the prescribed program or deviate from it. We present a sound technique for establishing programs as actual causes. We demonstrate the value of this approach by providing a causal analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure. Specifically, we analyze causes of authentication failures of a protocol that leverages a set of notaries to address concerns about trust-on-first-use of self-signed certificates.

Keywords: accountability, audit, security protocols, causality

1 Introduction

Accurate blame assignment for security violations is essential in a wide range of settings. For example, protocols for authentication and key exchange \cite{20}, electronic voting \cite{20}, auctions \cite{28}, and secure multiparty computation (in the semi-honest model) \cite{11} 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
blamed and appropriately punished is important to deter agents from committing future violations. Indeed the importance of blame assignment as a building block for accountability in information systems has been recognized in prior work \cite{1,13,21,23,27,33}. This paper focuses on the challenge of blame-assignment for security violations. Our thesis is that actual causation (i.e., identifying which agents’ deviations caused a specific violation) is a useful building block for blame-assignment in decentralized multi-agent systems, including but not limited to security protocols and ceremonies \cite{5}.

Actual causation has been studied extensively in philosophy, law, and computer science where 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 \cite{29}.) Notably, Hume \cite{18} 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, \cite{3,16,24,25,29,35}). This concept has generated significant interest in philosophy and law in part because of its connection with issues of moral and legal responsibility (see \cite{17,26}). Indeed that is also why we view actual causation as a useful building block for accountability in security settings.

At the same time, 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. Let us consider a simple protocol example. In the movie Flight \cite{10}, 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—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 according to a prescribed protocol.

This example is useful to illustrate several key ideas about blame-assignment that influence the formal development in this paper. First, it 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). Second, the example motivates us to hold accountable agents who exercise their choice to execute a deviant program that actually caused a violation. Thus, we focus on formalizing programs (executed by agents) as actual causes of violations rather than individual events as formalized
in prior work. Finally, it illustrates the importance of the actual interactions among agents for a violation to occur in a decentralized multi-agent system with non-deterministic execution semantics. For example, had the maintenance staff replaced the faulty component before take-off, the plane would not have crashed. The actual interaction structure thus plays a significant role in our formalization unlike prior work on actual causation [14, 16].

The central contributions of this paper are the following: We initiate a formal study of programs as actual causes. Specifically, we define what it means for a set of programs to be an actual cause of a violation when they run concurrently with a set of other programs. We also present a sound technique for establishing programs as actual causes building on prior work on proving security properties of protocols [4, 8]. We demonstrate the value of this approach by providing a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure.

In more detail, our formalization of programs as actual causes proceeds in three stages. The first stage captures two conditions. The Occurrence condition ensures that a safety property was violated on the log. The log records what programs actually executed and how they interacted. This model of a log is appropriate in settings where executed programs are available for analysis during forensics. The Necessity condition ensures that in a counterfactual scenario where all actually executed programs are replaced by the corresponding norms (i.e., the protocol specified ideal programs), the violation goes away on all resulting traces.

The second stage identifies a set $X$ of the actual programs as the suspected cause by requiring three conditions. The Closure condition ensures that every program that interacted with programs in $X$ on the log is also in $X$. The Sufficiency condition requires that all traces resulting from the execution of $X$ in conjunction with the norms for the other programs restores the violation so long as the programs in $X$ interact in a way consistent with the log. Finally, the Minimality condition requires that no subset of $X$ satisfies Closure and Sufficiency.

The third stage further separates the set $X$ into $X_d$ (the actual causes) and $X_p$ (the progress enablers). Informally, the actual values sent by the programs in $X_p$ are irrelevant to the violation—they merely serve to enable the progress of other programs that is essential for the violation. Roughly, this idea is formalized by replacing messages received by $X_d$ from $X_p$ by dummy values. An additional Sufficiency’ condition requires that the violation is restored under this transformation so long as the programs in $X_d$ interact in a way consistent with the log. A Minimality’ condition requires that no subset of $X_d$ satisfies Sufficiency’.

Our nearest neighbor is a treatment of actual causation by Halpern and Pearl [14, 15]. There are analogies between their conditions of Occurrence, Necessity, Sufficiency, and Minimality and our corresponding conditions in the first two stages. However, there are some important points of difference. First, in our non-deterministic, interactive program setting, it is critical to ensure that all interacting agents are identified in the Closure condition and the actual interactions of the suspected causes are held fixed as we move from Occurrence to the Sufficiency condition. There is no analogous concept in the work of Halpern.
and Pearl. Second, we exploit special characteristics of our security settings to simplify the definition. Specifically, in the Necessity condition when considering counterfactuals (alternative hypothetical scenarios in which the violation does not occur), it is essential to restrict the set of hypothetical scenarios to avoid counter-intuitive cause determinations. Halpern [14] recognizes this problem and considers only “more normal worlds” in his definition. While it may be challenging to figure out how to rank worlds according to normality in certain application domains, our insight is that the protocol-specified prescribed programs provide a natural basis for constructing such a ranking function. Finally, our third stage enables us to separate out the programs in $X_d$ whose information flows are causally related to the violation from the programs in $X_p$ who are just progress enablers but do not contribute relevant information flows. This distinction is important in security settings but is not considered by Halpern and Pearl.

The rest of the paper is organized as follows. Section 2 describes a representative example which we use throughout the paper to explain important concepts. Section 3 gives formal definitions for programs as actual causes of security violations. We apply the cause analysis to the running example in Section 4. We survey additional related work in Section 5 and conclude in Section 6.

2 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.

Security protocol. Consider an authentication protocol in which a user (User) authenticates to a server (Server) using a pre-shared password over an adversarial network. User sends its user-id to Server and obtains a key signed by Server. However, User would need inputs from additional sources when Server sends its public key for the first time in a protocol session to verify that the key is indeed bound to Server’s identity. In particular, User can verify the key by contacting multiple notaries in the spirit of Perspectives [34]. For simplicity, we assume User verifies Server’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.

The prescribed programs for Server, User and Notary impose certain requirements on their behavior: (i) Server stores User’s password in a hashed form in a secure memory location (ii) User requests access to the account by sending an encryption of the password (along with its identity and a timestamp) to Server after verifying Server’s public key with all notaries. User verifies that the majority of the notaries attested to the correctness of the key (iii) The notaries retrieve the key from their databases and attest the key correctly (iv) Server decrypts and computes the hashed value of the password (v) Server then 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 Server grants access to the account to User.
Security property. The prescribed programs in our example aim to achieve the property that only the user who created the account and password gains access to the account.

Compromised Notaries Attack. We describe an attack scenario and use it to illustrate nuances in formalizing programs as actual causes: User and Server execute norms. User sends an access request to Server. Adversary intercepts the message and sends a public key to User pretending to be Server. User checks with Notary1, Notary2 and Notary3 who falsely verify Adversary’s public key to be Server’s key. Consequently, User sends the password to Adversary. Adversary then initiates a protocol with Server and gains access to User’s account.

First, notice that the deviants in this example are Adversary, Notary1, Notary2 and Notary3. Analogous to Hume’s notion of counterfactual-based causation, our definition includes a necessity condition that captures the idea that there exists a hypothetical scenario in which the violation goes away. Specifically, we require that when Adversary performs no actions and the notaries, User, and Server follow their protocol-specified programs (norms), the violation goes away.

Second, notice that the deviant programs are not sufficient for the violation to occur without the involvement of User and Server. 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. 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 be found to be a cause. In addition, note that any strict superset of the set {Adversary, Notary1, Notary2, Notary3, User, Server} while being sufficient for the violation contains redundant programs. We will thus impose a minimality requirement on the set of actual causes to remove redundant programs from it.

Third, another nuance in defining the notion of sufficiency is to constrain the interactions among the programs in the actual cause set in a manner that is consistent with their actual interaction as recorded on the log. This constraint on interactions is quite subtle to define: if the constraint is too weak then the violation does not reappear in all traces thus breaking the sufficiency condition; 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 executes the same prefix in sufficiency as it does on the log. This notion does not work because for some violations to occur the order of interactions among threads is important. A notion that is too strong is to require matching of the total order of execution across all threads. We present a formal notion of log consistency in the next section that balances these competing concerns.

Finally, note that while all three notaries’ actions are required for the sequential User 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 programs who are progress enablers from those who provide information that cause the violation is useful for fine-grained causal determination. This observation motivates the final piece of our formal definition. It also helps
us identify independent causes, i.e., different sets of programs that independently caused the violation. In our example, we get three different independent causes depending on which notary is treated as a progress enabler.

3 Actual Cause Definition

We introduce our model in Section 3.1, define auxiliary notions (like log consistency) in Section 3.2, and present the formal definition of programs as actual causes in Section 3.3.

3.1 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 may do one of the following: execute a primitive function $\zeta$ on a term $t$ (written $\zeta(t)$), sequentially compose two expressions (written $x = e_1; e_2$), check for the truth of a boolean expression (written $\text{assert}(e)$) and send or receive a message (written $\text{send}(t)$ and $\text{recv}()$, respectively).

Expressions $e ::= b : \zeta(t) \mid x = e_1; e_2 \mid b : \text{assert}(e) \mid b : \text{send}(t) \mid b : \text{recv}()$

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). Expression $\text{send}(t)$ always returns 0 to its continuation.

Primitive functions $\zeta$ 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. We assume that there is an identity primitive function that returns its argument. This function applied to $t$ is also written $t$. If the expression $e$ in $\text{assert}(e)$ evaluates to a non-true value, then its containing thread gets stuck forever. Each atomic expression is labelled with a unique line number, written $b$. Line numbers help define logs later. We omit line numbers when they are not relevant. 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 $j$.

$m = \text{recv}();$ // receive message, bind to $m$
$n = \text{new}();$ // generate nonce, bind to $n$
$\text{send}(j, (m, n));$ // 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 auxiliary definitions such as the log, synchronization and log consistency must be modified appropriately.

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 $e$, a local stack and a local store. A configuration $C = T_1, \ldots, T_n$ models the threads $T_1, \ldots, T_n$ executing concurrently. Our reduction relation is written $C \rightarrow C'$ and defined in the standard way by interleaving small steps of individual threads (the reduction relation is parametrized by a semantics of primitive functions $\zeta_i$). 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 $C^r \rightarrow C'$. A label is either the identifier of a thread $i$ paired with a line number $b$, written $(i, b)$, representing an internal reduction in thread $i$ at line number $b$, or a tuple $(\langle i_s, b_s \rangle, \langle i_r, b_r \rangle)$, representing a synchronization between a send at line number $b_s$ in thread $i_s$ and a recv at line number $b_r$ in thread $i_r$. Labels $(i, b)$ are called local labels and labels $(\langle i_s, b_s \rangle, \langle i_r, b_r \rangle)$ are called synchronization labels.

An initial configuration can be described by a triple $(I, E, \Sigma)$, where $I$ is a finite set of thread identifiers, $E : I \rightarrow$ Expressions and $\Sigma : I \rightarrow$ Stores. This defines an initial configuration of $|I|$ threads with identifiers in $I$, where each thread contains an empty stack, thread $i$ contains the expression $E(i)$ and thread $i$ contains the store $\Sigma(i)$. In the sequel, we identify the triple $(I, E, \Sigma)$ with the configuration defined by it. We also use a configuration’s identifiers to refer to its threads.

Definition 1 (Trace). Given an initial configuration $C_0 = (I, E, \Sigma)$, a trace is a sequence of valid labelled reductions, $C_0^r_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_m$.

3.2 Logs and their projections

To define actual causation, we find it convenient to introduce the notion of the log of a trace, which is just the sequence of labels on the trace.

Definition 2 (Log). Given a trace $t = C_0^r_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_m$, the log of the trace $t$, $\log(t)$, is the sequence $r_0, \ldots, r_{m-1}$.

The letter $l$ denotes logs. We need a few more straightforward definitions on logs in order to define actual causation. In the sequel, $X, Y, Z$ denote sets of thread identifiers. A sublog of $l = r_0, \ldots, r_m$ is a subsequence of $r_0, \ldots, r_m$.

Definition 3 (Closed set of thread identifiers). A set of thread identifiers $X$ is said to be closed on log $l$ if for every synchronization label $(\langle i_s, b_s \rangle, \langle i_r, b_r \rangle) \in l$, $i_s \in X$ if and only if $i_r \in X$. 
Intuitively, $X$ is closed on $l$ if threads in $X$ synchronize only amongst themselves according to $l$.

**Definition 4 (Local projection of a log).** The local projection of a log $l$ with respect to a thread identifier $i$, written $l \downarrow_i$, is the sublog of $l$ containing all local labels whose thread identifier is $i$. Formally,

\[
\begin{align*}
\bullet \downarrow_i &= \bullet \\
((i, b) :: l) \downarrow_i &= (i, b) :: (l \downarrow_i) \\
((j, b) :: l) \downarrow_i &= l \downarrow_i \quad \text{if } i \neq j \\
((\langle i_s, b_s \rangle, \langle i_r, b_r \rangle) :: l) \downarrow_i &= l \downarrow_i 
\end{align*}
\]

**Definition 5 (Synchronization projection of a log).** The synchronization projection of a log $l$ with respect to a set of thread identifiers $X$, written $l|_X$, is the sublog of $l$ containing synchronization labels from $l$ both of whose thread identifiers are in $X$. Formally,

\[
\begin{align*}
\bullet|_X &= \bullet \\
((j, b) :: l)|_X &= l|_X \\
((\langle i_s, b_s \rangle, \langle i_r, b_r \rangle) :: l)|_X &= (\langle i_s, b_s \rangle, \langle i_r, b_r \rangle) :: (l|_X) \quad \text{if } i_s, i_r \in X \\
((\langle i_s, b_s \rangle, \langle i_r, b_r \rangle) :: l)|_X &= l|_X \quad \text{if } i_s \notin X \lor i_r \notin X 
\end{align*}
\]

A log $l'$ is called consistent with a log $l$ relative to a set of thread identifiers $X$ if $X$ is closed on $l'$ and the local and synchronization labels of threads in $X$ are identical (and identically ordered) in $l$ and $l'$. This is formalized below.

**Definition 6 (Log consistency).** A log $l'$ is said to be consistent with a log $l$ relative to a set of thread identifiers $X$ if the following hold:

1. $X$ is closed on $l'$
2. $\forall i \in X. l' \downarrow_i = l \downarrow_i$
3. $l'|_X = l|_X$

### 3.3 Programs as Actual Causes

Our goal is to define actual causes of the violation of an expected security property. By property in this paper we mean any safety property of labelled traces that is closed under permutation of reductions that are unrelated to each other in Lamport’s “happens-before” relation [22]. We use $\varphi_V$ to denote the complement of such a property (i.e., $\varphi_V$ is the set of property-violating traces).

Consider a trace $t$ starting from the initial configuration $C_0 = \langle I, A, \Sigma \rangle$. Suppose $t \in \varphi_V$, so this trace violates the safety property $\neg \varphi_V$. Our definition of actual causation identifies a subset of the threads $I$ as the cause of the violation. To do this, we employ a set of hypothetical counterfactual scenarios, in which subsets of the programs $A(i), i \in I$ are replaced by norms or the correct, prescribed programs. Consequently, we assume that we are provided a second function $N : I \rightarrow \text{Expressions}$ such that $N(i)$ is the program that ideally should
have been executing in the thread $i$. Note that for some $i$, $A(i)$ and $N(i)$ may be equal, but $A(i)$ may still have contributed to the violation in collaboration with other threads $j$ for which $A(j) \neq N(j)$. For each $i$, we call $N(i)$ the norm for thread $i$ and $A(i)$ the “actual” for thread $i$.

We formalize a violation and the corresponding norms in a violation structure, to which our definition of actual causation applies. We impose two conditions on a violation structure. First, there must actually be a violation, else looking for causes is meaningless. We call this condition occurrence. Second, 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 define causes is reasonable.

**Definition 7 (Violation structure).** A violation structure is a tuple $V = (I, A, \Sigma, N, \varphi_V, t)$ such that $t$ is a trace starting from $(I, A, \Sigma)$ and the following two conditions hold:

1. *(Occurrence)* $t \in \varphi_V$.
2. *(Necessity)* For any trace $t'$ starting from the initial configuration $(I, N, \Sigma)$, it is the case that $t' \not\in \varphi_V$.

When considering counterfactuals, we often replace the actuals in a subset $Y$ of the threads $I$ with their norms. The following definition captures the resulting initial configuration.

**Definition 8 (Normification).** Given a violation structure $V = (I, A, \Sigma, N, \varphi_V, t)$ and a partition $(X, Y)$ of $I$, we define the normified initial configuration $\text{norm}(V, X, Y) = (I, E, \Sigma)$, where

$$E(i) = \begin{cases} N(i) & \text{if } i \in Y \\ A(i) & \text{if } i \in X \end{cases}$$

Armed with these definitions, we are now in a position to formally define programs as actual causes. Our definition applies in two phases. The first phase identifies suspected causes. Technically, a suspected cause is a minimal, closed set of threads that can account for the violation, even if all other threads are replaced by norms. In the second phase, we refine this set into actual causes and progress enablers. The latter contribute only indirectly to the cause by enabling the actual causes to make progress; the exact values transmitted by progress enablers are irrelevant.

Our Phase 1 definition below determines suspected causes. It contains two conditions. The sufficiency condition tests that the suspected causes, when combined with norms for the remaining threads, suffice to recreate the violation. Technically, we consider all traces from a normified counterfactual, in which the candidate suspected causes (called $X$ in the definition) follow the same sequence of reductions as in the original trace. A key criteria is that $X$ be closed on the violating trace’s log. This is important, because any thread that communicates with a suspected cause may have at the least enabled progress of the latter and, hence, contributed to the violation. The minimality condition tests that the identified suspected causes contain no redundant threads.
Definition 9 (Suspected Cause of Violation: Phase 1). Let \( V = (I, A, \Sigma, N', \varphi_V, t) \) be a violation structure and \( l = \log(t) \). We say that \( X \subseteq I \) is a suspected cause of the violation \( V \) if the following hold:

1. **(Closure)** \( X \) is closed on \( l \).
2. **(Sufficiency)** Let \( Y = I \setminus X \) and \( C'_0 = \text{norm}(V, X, Y) \). Let \( T \) be the set of traces starting from \( C'_0 \) that are log-consistent with \( l \) relative to \( X \). Then, \( T \) is non-empty and \( T \subseteq \varphi_V \).
3. **(Minimality)** No proper subset \( X' \) of \( X \) satisfies conditions 1 and 2.

The Phase 1 definition above identifies a minimal set \( X \) of threads, which is sufficient to cause the violation and does not interact with other threads (called \( Y \)). In the Phase 2 definition below, we further partition \( X \) into \( X_d \) (actual cause) and \( X_p \) (progress enablers) such that the threads in \( X_p \) contribute only towards the progress of other threads that cause the violation. In other words, the set \( X_p \) contains all threads whose actual transmitted values are irrelevant.

Briefly, here’s how our Phase 2 definition works. We first pick a candidate set \( X_d \subseteq X \) (where \( X \) is the suspected cause set identified in Phase 1) and define \( X_p = X \setminus X_d \). We consider counterfactual traces obtained from initial configurations in which threads from \( X_p \) (the hypothesized progress enablers) are completely dropped and, instead, any inputs that threads of \( X_p \) are replaced by arbitrary dummy values and, additionally, the synchronizations within \( X_d \) are the same as in the original violating trace. If a violation appears in all such counterfactual traces, then the partition of \( X \) into \( X_d \) and \( X_p \) is a good candidate. Of all such good candidates, we choose those with minimal \( X_d \) (or, equivalently, maximal \( X_p \)).

The key technical difficulty in writing this definition is replacing values communicated from \( X_p \) to \( X_d \) with arbitrary dummy values. While there are many ways to do this, we choose a simple method: We syntactically transform initial expressions of threads in \( X_d \), replacing every \( \text{recv()} \), which synchronized with an expression in \( X_p \), with a dummy value. Since our communication model is synchronous, we must also erase all \( \text{send()} \) expressions from threads in \( X_d \), if the recipient was in \( X_p \). The following definition formalizes this idea. It defines a new initial configuration obtained by replacement with dummy values in \( X_d \), removal of \( X_p \) and normification of threads outside \( X_d \) and \( X_p \). The function \( f \) supplies dummy values for use in replacement. In the Phase 2 definition, we quantify universally over this function.

Definition 10 (Dummifying transformation). Let \( V = (I, A, \Sigma, N, \varphi_V, t) \) be a violation structure, \( X_d, X_p, \) be disjoint subsets of \( I \), \( l = \log(t) \) and \( f : I \times \text{LineNumbers} \rightarrow \text{Terms} \). The dummifying transform \( \text{dummify}(V, X_d, X_p, f) \) is the initial configuration \( (I \setminus X_p, \mathcal{E}, \Sigma) \), where \( \mathcal{E} \) is defined as follows:

- For \( i \in (I \setminus X_p) \setminus X_d \), \( \mathcal{E}(i) = N(i) \).
- For \( i \in X_d \), \( \mathcal{E}(i) \) is \( A(i) \) modified as follows:
  - If \( \langle j, b_r \rangle, \langle i, b_r \rangle \rangle \in l \) and \( j \in X_p \), then replace \( b_r : \text{recv()} \) in \( A(i) \) with \( b_r : f(i, b_r) \).
Definition 11 (Actual Cause of Violation: Phase 2). Let \( V = (I, A, \Sigma, N, \varphi_V, t) \) be a violation structure and \( l = \log(t) \). Let \( X \) be a suspected cause of the violation \( V \) determined by Definition 9. We say that \( X_d \subseteq X \) is an actual cause of the violation \( V \) if the following hold:

1. (\textbf{Sufficiency'}) Let \( X_p = X \setminus X_d \). For every \( f \), if \( C_0' = \text{dummify}(V, X_d, X_p, f) \) and \( T \) is the set of traces starting from \( C_0' \) that are log-consistent with \( l \) relative to \( X_d \), then \( T \) is non-empty and \( T \subseteq \varphi_V \).
2. (\textbf{Minimality'}) No proper subset \( X' \) of \( X_d \) satisfies condition 1.

More than one minimal set \( X_d \) may satisfy the above Phase 2 definition for a given violation \( V \). Every such \( X_d \) is deemed an independent actual cause of the violation.

4 Causes of Authentication Failures

In this section, we model an instance of our running example based on passwords (Section 2) to demonstrate our definition of actual cause. As explained in Section 2, we consider a protocol session where User, Server 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 2 and provide a cause analysis using the definitions from Section 3.

4.1 Protocol Description

We consider our example protocol with six threads named \{Server, User, Adversary, Notary1, Notary2, Notary3\}. A relevant part of the norms for all these threads, except Adversary, are shown in Figure 1. Adversary’s norm is empty, because in an ideal world, Adversary should not participate. (In our appendix, we demonstrate the cause analysis with more threads which do not interact with these six threads and are filtered out by our Phase 1 definition. We also give the complete proof considering timestamps and the account creation phase.)

The norms in Figure 1 assume that User’s account (called acct in Server’s norm) has already been created and that User’s password, pwd is associated with User’s user id, uid. This association (in hashed form) is stored in Server’s local state at pointer mem. The norm for Server is to wait for a request from an entity, respond with its (Server’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 Server’s memory at mem. To grant access, Server adds an entry into a private access matrix, called P. (A separate server thread, not shown here, allows User to access its account if this entry exists in P.)
The norm for User is to send an access request to Server, wait for the server’s public key, verify that key with three notaries and then send its password pwd to Server, encrypted under Server’s public key. On receiving Server’s public key, User 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 User and the notaries that authenticates the notaries and ensures freshness of their responses. These details are included in our appendix.

Each notary has a private database of (public_key, principal) tuples. The notaries’ norms assume that this database has already been created correctly. When User 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. (In this example, we identify threads with principals.)

Notation. The norms in Figure 1 use several primitive functions. Enc(k, m) and Dec(k’, m) denote encryption and decryption of message m with key k and k’ respectively. Hash(m) generates the hash of term m. 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 send(i, m) and recv(j) expressions. As explained earlier, i and j are ignored during execution and the adversary, if present, may capture or inject any messages.

Security property. The security property, ¬φV, of interest to us is that if at time u, a thread k is given access to account a, then k owns a. This can be formalized by the following logical formula:

∀u, k, a. (a, k) ∈ P(u) ⇒ Acct.user(a, k, u) (1)

Here, P(u) is the state of the access control matrix P at time u and the predicate Acct.user(a, k, u) holds if the thread k owns account a at time u. We assume that User, not Adversary, owns account acct. (We use this logical formalization of the property in establishing the actual causes using our definition. Specifically, we use a program logic to establish the sufficiency and necessity conditions.)

4.2 Attack

As an illustration, we model the violation in the “Compromised Notaries Attack” of Section 2. In this attack scenario, User and Server execute norms. User sends an access request to Server which is intercepted by Adversary who sends its own key to User (pretending to be Server). User checks with the three notaries who falsely verify Adversary’s public key to be Server’s key. Consequently, User sends the password to Adversary. Adversary then initiates a protocol with Server and gains access to User’s account. In the property-violating trace, User and Server execute their norms from Figure 1 and the expressions executed by Adversary and the
The concrete trace we consider has the following log, \( I \):

\[
\begin{align*}
\langle \text{User}, 1 \rangle, & \langle \text{Adversary}, 1 \rangle, \\
\langle \text{Adversary}, 2 \rangle, & \langle \text{User}, 2 \rangle, \\
\langle \text{User}, 3 \rangle, & \langle \text{Notary1}, 1 \rangle, \\
\langle \text{User}, 4 \rangle, & \langle \text{Notary2}, 1 \rangle, \\
\langle \text{User}, 5 \rangle, & \langle \text{Notary3}, 1 \rangle, \\
\langle \text{Notary1}, 2 \rangle, & \langle \text{User}, 6 \rangle, \\
\langle \text{Notary2}, 2 \rangle, & \langle \text{User}, 7 \rangle, \\
\langle \text{Notary3}, 2 \rangle, & \langle \text{User}, 8 \rangle, \\
\langle \text{User}, 9 \rangle, & \langle \text{User}, 10 \rangle, \\
\langle \langle \text{User}, 11 \rangle, & \langle \text{Adversary}, 3 \rangle, \\
\langle \text{Adversary}, 4 \rangle, & \langle \langle \text{Adversary}, 5 \rangle, \langle \text{Server}, 1 \rangle, \\
\langle \text{Server}, 2 \rangle, & \langle \text{Adversary}, 6 \rangle, \\
\langle \text{Adversary}, 7 \rangle, & \langle \langle \text{Adversary}, 8 \rangle, \langle \text{Server}, 3 \rangle, \\
\langle \text{Server}, 4 \rangle, & \langle \text{Server}, 5 \rangle, \\
\langle \text{Server}, 6 \rangle, & \langle \text{Server}, 7 \rangle
\end{align*}
\]

three notaries are shown in Figure 2. Thus, the initial configuration has the programs \( \mathcal{N} \langle \text{User} \rangle, \mathcal{N} \langle \text{Server} \rangle, \mathcal{A} \langle \text{Adversary} \rangle, \mathcal{A} \langle \text{Notary1} \rangle, \mathcal{A} \langle \text{Notary2} \rangle, \mathcal{A} \langle \text{Notary3} \rangle \).

Fig. 1. Norms for all threads. Adversary’s norm is the trivial empty program.
Actual program $A(A(\text{Adversary}))$

1: \texttt{recv}($\text{User}$); //intercept access req from $\text{User}$
2: $s = \texttt{recv}($$\text{User}$, $\text{pubkey}_A)$; //send key to $\text{User}$
3: $s = \texttt{recv}($$\text{User}$); //pwd from $\text{User}$
4: \((uid, pwd, User) = \text{Dec}(\texttt{pvtkey}_A, s)$$); //decrypt pwd
5: \texttt{send}($\text{Server}$, $uid$); //access request to $\text{Server}$
6: $\texttt{pubkey} = \texttt{recv}($$\text{Server}$); //Receive $\text{Server}$’s public key
7: $t = \texttt{Enc}(\texttt{pubkey}, (uid, pwd, \text{Adversary}))$; //encrypt pwd
8: \texttt{send}($\text{Server}$, $t$); //pwd to $\text{Server}$

Actual programs $A(A(\text{Notary}1), A(\text{Notary}2), A(\text{Notary}3))$:

// $o$ denotes $\text{Notary}1$, $\text{Notary}2$ or $\text{Notary}3$
1: $\text{Enc}(\texttt{pubkey}_o, \texttt{pubkey}) = \texttt{recv}(j)$;
2: $\texttt{send}(j, \text{Sig}(\texttt{pvtkey}_o, (\texttt{pubkey}, \text{Server})))$

Fig. 2. Attack scenario 1: Deviants for $\text{Adversary}$ and $\text{Notary}$

At the end of this log, (acct, $\text{Adversary}$) occurs in the access control matrix $P$, but $\text{Adversary}$ does not own acct. Hence, this log corresponds to a violation of our security property. Importantly, even though only $\text{Adversary}$ and the notaries deviate from their norms, this violation cannot happen without participation from $\text{User}$ and $\text{Server}$. Moreover, if any two of the three notaries had deviated from their norm, the violation would have still happened. Consequently, we may expect three independent causes in this example: \{Adversary, User, Server, Notary1, Notary2\}, \{Adversary, User, Server, Notary1, Notary3\}, and \{Adversary, User, Server, Notary2, Notary3\}. The following theorem states that our definitions determine exactly these three independent causes.

**Theorem 1.** Let $I = \{\text{User, Server, Adversary, Notary1, Notary2, Notary3}\}$, $l$ be the log above, $t$ be a trace with $\log(t) = l$ and $A, N$ and $\Sigma$ be as described above. Let $V = (I, A, \Sigma, N, \varphi_V, t)$. Then, Definition 11 determines three possible values for the actual cause $X_d$ of violation: \{Adversary, User, Server, Notary1, Notary2\}, \{Adversary, User, Server, Notary1, Notary3\}, and \{Adversary, User, Server, Notary2, Notary3\}.

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that $V$ is a violation structure and that our Phase 1 and Phase 2 definitions yield exactly the three values for $X_d$ mentioned in the theorem. To check that $V$ is a violation structure, we must verify the occurrence and necessity conditions from Definition 7. For occurrence, we must show that $t \in \varphi_V$. This is clear because at the end of the trace, (acct, $\text{Adversary}$) $\in P$, but by our assumption, User, not $\text{Adversary}$ owns acct. For necessity, we show that any trace starting from $\langle I, N, \Sigma \rangle$ cannot be in $\varphi_V$, i.e., any such trace has the invariant $\text{Security property}$. To prove this, we can use any sound program logic. In particular, we use the logic of Garg et al. $\text{[8]}$. Our proof, in fact, shows the stronger property that $\text{Security property}$ is an invariant even when the six
Programs as Actual Causes

15

norms execute concurrently with any number of standard Dolev-Yao adversaries (we assume that encryption is non-malleable). This is just a standard proof of protocol correctness with a program logic, so we defer its details to our appendix.

**Phase 1:** Phase 1 (Definition 9) determines $X = I$ and $Y = I \setminus X = \{\}$. This is easily proved: $I$ contains a single connected component in the synchronization graph induced by $l$, so closure of $X$ on $l$ forces $X = I$ or $X = \{\}$. If $X = \{\}$, then $Y = I$, so in the sufficiency condition, we would have to produce a violation using norms for all six threads, which we proved impossible in the necessity condition above. Hence, $X = I$.

**Phase 2:** Phase 2 (Definition 11) determines three independent actual causes for $X_d$: \{Adversary, User, Server, Notary1, Notary2\}, \{Adversary, User, Server, Notary1, Notary3\}, and \{Adversary, User, Server, Notary2, Notary3\}. These are symmetric, so we only explain why $X_d = \{Adversary, User, Server, Notary1, Notary2\}$ satisfies Definition 11. We show that (a) This set $X_d$ satisfies sufficiency’, and (b) No proper subset of $X_d$ satisfies sufficiency’ (minimality’).

We start with (a). By definition, $X_p = X \setminus X_d = \{Notary3\}$. Fix any dummifying function $f$. We must show two things. First, there is a trace from $\text{dummify}(V, X_d, X_p, f)$, whose log is consistent with $l$ relative to $X_d$. Second, any such trace is in $\varphi_V$. The first statement is easily established by mimicking the execution in $l$ starting from $\text{dummify}(V, X_d, X_p, f)$. The only potential issue is that on line 8, $\mathcal{A}(\text{User})$ (which equals $\mathcal{N}(\text{User})$ from Figure II) synchronizes with Notary3 according to $l$, but Notary3 does not exist in $\text{dummify}(V, X_d, X_p, f)$. However, in $\text{dummify}(V, X_d, X_p, f)$, the $\text{recv}()$ on line 8 in $\mathcal{A}(\text{User})$ is replaced with a dummy value, so the execution from $\text{dummify}(V, X_d, X_p, f)$ progresses. Subsequently, the majority check on line 9 succeeds as in $l$, because two of three notaries (Notary1 and Notary2) still attest the Adversary’s key.

The second statement — that every trace starting from $\text{dummify}(V, X_d, X_p, f)$, whose log is consistent with $l$ relative to $X_d$, is in $\varphi_V$ — is the most non-obvious part of our proof. This statement quantifies over a select set of traces, but not all, so we cannot directly use the program logic. Instead, we use the program logic to establish certain invariants and combine those invariants with our knowledge of $l$. Fix a trace $t'$ with log $l'$. Assume $l'$ consistent with $l$ relative to $X_d$. We show that $t' \in \varphi_V$ as follows.

1. Since $l'$ is consistent with $l$ relative to $X_d$, Server must execute line 9 of its program $\mathcal{N}(\text{Server})$ in $t'$. Consequently, at some point of time, the access control matrix $P$ contains $(acct, J)$ for some $J$.
2. We prove an invariant of $\mathcal{N}(\text{Server})$ that if it writes $(x, J)$ to $P$, then $J$ is the third component of a tuple obtained by decrypting a message received on line 3.
3. Since $l'$ is consistent with $l$ relative to $X_d$, and on $l$ (Server, 3) synchronizes with (Adversary, 8), $J$ must be the third component of an encrypted message sent on line 8 of $\mathcal{A}(\text{Adversary})$. 
4. We prove an invariant of $A(\text{Adversary})$ that the third component of the message sent on line 8 is exactly the term "Adversary". (This is easy to establish, as the term "Adversary" is hardcoded on line 7.) Hence, $J = \text{Adversary}$. 

5. This immediately implies that $t' \in \varphi_V$ since $(\text{acct}, \text{Adversary}) \in P$, but by assumption, User, not Adversary owns acct.

Last, we prove (b) — that no proper subset $X'_d$ of $X_d$ satisfies sufficiency'. Pick any $X'_d \subseteq X_d$ such that $X'_d$ satisfies sufficiency'. We show that $X'_d = X_d$. Let $X'_p = X \setminus X'_d$. Observe that if Server $\not\in X'_d$, then Server is not in dummify$(V, X'_d, X'_p, f)$ and, hence, on any counterfactual trace cannot write to $P$, thus precluding a violation. Therefore, Server $\in X'_d$. By sufficiency', for any $f$, the log $l'$ of at least one trace $t'$ of dummify$(V, X'_d, X'_p, f)$ must be consistent with $l$ relative to $X'_d$. This means that in $t'$, the assertion on line 5 of $N$(Server) must succeed and, hence, on line 4, the correct password $pwd$ must be received by Server, independent of $f$. This immediately implies that Adversary (which sends that password on $l$) must be in $X'_d$, else some dummified executions of Server will have the wrong password and the assertion on line 5 will fail.

Extending this logic further, we now observe that because Adversary forwards a password received from User (line 3 of $A(\text{Adversary})$) to Server, User $\in X'_d$ (otherwise, some dummifications of line 3 of $A(\text{Adversary})$ will result in the wrong password being sent to Server, a contradiction). Since User $\in X'_d$ and $l'$ must be consistent with $l$ relative to $X'_d$, the majority check on line 9 of $N$(User) must also succeed. This means that at least two of $\{\text{Notary}_1, \text{Notary}_2, \text{Notary}_3\}$ must be in $X'_d$, else the dummification of lines 6–8 of $N$(User) will cause the assertion check to fail for some $f$. Since $X'_d \subseteq X_d$, these two threads must be $\{\text{Notary}_1, \text{Notary}_2\}$. At this point we have established that each of $\{\text{Adversary}, \text{User}, \text{Server}, \text{Notary}_1, \text{Notary}_2\}$ is in $X'_d$. Hence, $X'_d = X_d$.

5 Related Work

Currently, there are multiple proposals for providing accountability in decentralized multi-agent systems [1,2,6,7,12,13,19,21,32].

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 blame agents.

On the other hand, prior work on actual causation in analytical philosophy and AI has considered counterfactual based causation in detail [3,15,16,25,29,35]. 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.
5.1 Accountability

Küsters et al [21] 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 \( \pi \) of a protocol \( P \), where the verdict blames agents. Instances of \( P \) are non-deterministic processes with concurrently executing programs. 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 specific accountability property. In contrast, our cause definition provides 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. In Küsters et al.’s framework, it is desirable that a judge give “minimal” verdicts and not blame agents who deviate in a manner that is not relevant to the violation. However, there is no formal definition for a verdict being minimal. We believe that using our cause definition as the basis for accountability constraints would ensure the minimality of verdicts.

Backes et al [1] define accountability as the ability to show evidence when an agent deviates. The authors analyze a contract signing protocol using protocol composition logic and prove certain properties about the protocol. 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. where the property associated with the violating trace is an example of an accountability constraint.

Feigenbaum et al [6,7] also propose a definition of accountability that focuses on linking a violation to punishment. They use Halpern and Pearl’s definition [15, 16] 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. We think that the use of causality in that setting is an apt choice. 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 regard programs as causes of violations while they establish a causal chain between violation and punishment events. While the event-based view is suitable for their work, we discuss in Section 4 why that view is not suitable for our purposes.

Jagadeesan et al [19] provide a definition of accountability in the context of authorization in distributed systems. Their analysis models an auditor as an honest agent of the protocol and uses game-based logics in order to study the tradeoffs between the requirements for honest agents and the audit protocols. The auditor considers the actions of agents in the protocol and can assign blame
to the agents deviating from the protocol. In contrast, in our work, we consider a trace-based model where a violation is detected on a finite trace and reason about an agent’s program being a cause of the violation (not just a deviant).

Haeberlen et al [13] provide a definition of accountability in distributed systems. Their system, PeerReview, maintains a record of all nodes’ actions and provides non-repudiable evidence when a node deviates from its local protocol. The ‘violation’ occurs when a node does not respond to a message as expected by the protocol or sends a message not prescribed by the protocol. The motivation of this work differs from ours as we try to find causes of a global violation of a protocol specification. We differentiate between deviance, which occurs when an agent does not follow its locally specified program and a global violation, which occurs when a protocol property is not satisfied.

5.2 Causation for blame assignment

The work by Barth et al [2] provides a definition of accountability that uses the much coarser notion of Lamport causality. Lamport causality is related to closure in 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 [12] use causation to link deviation of local specification by individual components of the system with the violation of the global specification, i.e., system failure. They treat components as black boxes which have local contracts specifying the desired behavior and model each component as a labelled transition system. The authors consider a trace-based model and reconstruct traces by replacing prefixes of traces containing actions of faulty components by outputs which would have arisen had the components followed their contracts. An important distinction from our formalization is that they consider alternative scenarios where inputs and outputs of components can be constrained, without taking into effect all interactions. It is not possible to construct alternate traces where faulty components interact with other components in a manner different from that on the log. Preserving the interaction structure on the log is crucial for our definitions.

Wang et al [32] describe a counterexample to Gössler et al’s work [12] where all causes are not found because of not being able to completely capture the effect of one component’s behavior on another’s. Wang et al [32] propose an approach similar to Gössler et al’s for determining blameworthy components in fault diagnosis. However, they differ in the manner of reconstructing traces and finding causes. Here too, alternate traces cannot be generated since components are considered black box.

6 Conclusion

We have presented a first attempt at defining what it means for a set of programs 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 practical applications.

References

1. Backes, M., Datta, A., Derek, A., Mitchell, J.C., Turuani, M.: Compositional analysis of contract-signing protocols. Theor. Comput. Sci. 367(1-2), 33–56 (2006)
2. Barth, A., Mitchell, J.C., Datta, A., Sundaram, S.: Privacy and utility in business processes. In: CSF. pp. 279–294 (2007)
3. Collins, J., Hall, N., Paul, L.A.: Causation and Counterfactuals. MIT Press (2004)
4. Datta, A., Derek, A., Mitchell, J.C., Pavlovic, D.: A derivation system and compositional logic for security protocols. Journal of Computer Security 13(3), 423–482 (2005)
5. Ellison, C.: Ceremony design and analysis. http://eprint.iacr.org/2007/399.pdf
6. Feigenbaum, J., Hendler, J.A., Jaggard, A.D., Weitzner, D.J., Wright, R.N.: Accountability and deterrence in online life. In: Proceedings of the 3rd International Web Science Conference. pp. 7:1–7:7. WebSci ’11, ACM, NY, USA (2011)
7. Feigenbaum, J., Jaggard, A.D., Wright, R.N.: Towards a formal model of accountability. In: Proceedings of the 2011 workshop on New security paradigms workshop. pp. 45–56. NSPW ’11, ACM, New York, NY, USA (2011)
8. Garg, D., Franklin, J., Kaynar, D., Datta, A.: Compositional system security with interface-confined adversaries. Electron. Notes Theor. Comput. Sci. 265, 49–71 (Sep 2010)
9. Garg, D., Franklin, J., Kaynar, D., Datta, A.: Compositional system security with interface-confined adversaries (technical report. Tech. Rep. CMU-CyLab-10-004, Carnegie Mellon University (Feb 2010), http://www.mpi-sws.org/~dg/papers/cmu-cylab-10-004.pdf
10. Gatins, J.: Flight (2012), http://www.imdb.com/title/tt1907668/?
11. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 218–229. STOC ’87, ACM, New York, NY, USA (1987)
12. G"ossler, G., Le M"etayer, D., Raclet, J.B.: Causality analysis in contract violation. In: Proceedings of the First International Conference on Runtime Verification. pp. 270–284. RV’10, Springer-Verlag, Berlin, Heidelberg (2010)
13. Haeberlen, A., Kouznetsov, P., Drschel, P.: Peerreview: practical accountability for distributed systems. In: SOSP. pp. 175–188 (2007)
14. Halpern, J.Y.: Defaults and Normality in Causal Structures. Artificial Intelligence 30, 198–208 (2008), http://arxiv.org/abs/0806.2140
15. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach: part i: causes. In: Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence. pp. 194–202. UAI’01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
16. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach. part i: Causes. British Journal for the Philosophy of Science 56(4), 843–887 (2005)
17. Hart, H.L.A., Honore, T.: Causation in the Law (Second Edition). Oxford University Press, Oxford, UK (1985)
18. Hume, D.: An Enquiry Concerning Human Understanding, Reprinted Open Court Press, LaSalle, IL, 1958 (1748)
19. Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: Towards a theory of accountability and audit. In: ESORICS, pp. 152–167 (2009)
20. Kaufman, C., Perlman, R., Speciner, M.: Network security: private communication in a public world. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1995)
21. Küsters, R., Truderung, T., Vogt, A.: Accountability: Definition and Relationship to Verifiability. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010), pp. 526–535. ACM Press (2010)
22. Lamport, L.: Ti clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (July 1978)
23. Lampson, B.: Computer security in the real world. Computer 37(6), 37 – 46 (June 2004)
24. Lewis, D.: Causation. Journal of Philosophy 70(17), 556–567 (1973)
25. Mackie, J.L.: Causes and Conditions. American Philosophical Quarterly 2(4), 245–264 (1965)
26. Moore, M.S.: Causation and Responsibility: An Essay in Law, Morals and Metaphysics. Oxford University Press (2009)
27. Nissenbaum, H.: Accountability in a computerized society. Science and Engineering Ethics 2(1), 25–42 (1996)
28. Parkes, D.C., Rabin, M.O., Shieber, S.M., Thorpe, C.: Practical secrecy-preserving, verifiably correct and trustworthy auctions. Electron. Commer. Rec. Appl. 7(3), 294–312 (Nov 2008)
29. Pearl, J.: Causality: models, reasoning, and inference. Cambridge University Press, New York, NY, USA (2000)
30. Rivest, R.L., Smith, W.D.: Three voting protocols: Threeballot, vav, and twin. In: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology. pp. 16–16. EVT'07, USENIX Association, Berkeley, CA, USA (2007)
31. on Tort Law, E.G.: Principles of European Tort Law: Text and Commentary. Springer (2005).
32. Wang, S., Ayoub, A., Ivanov, R., Sokolsky, O., Lee, I.: Contract-based blame assignment by trace analysis. In: Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems. ACM, NY, USA (2013)
33. Weitzner, D.J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J., Sussman, G.J.: Information accountability. Commun. ACM 51(6), 82–87 (Jun 2008)
34. Wendlandt, D., Andersen, D.G., Perrig, A.: Perspectives: improving ssh-style host authentication with multi-path probing. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference. USENIX Association, CA, USA (2008)
35. Wright, R.: Causation in tort law. California Law Review 73 pp. 1735–1828 (1985)
Appendix

6.1 Operational Semantics

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

$$
\sigma \triangleright T \mapsto \sigma' \triangleright T'
$$

**eval** $t'$$

$$
\sigma \triangleright I; (x.e) :: K; t \mapsto \sigma \triangleright I; K; e[t'/x]
$$

**red-term**

$$
\sigma; I \triangleright e \mapsto \sigma' ; I \triangleright t
\text{eval } t'
\sigma \triangleright I; (x.e_1) :: K; b : \zeta(e)^{(t,b)} \mapsto \sigma' \triangleright I; K; e_1[t'/x]
$$

**red-act**

$$
\sigma \triangleright I; K; \text{assert}(\text{bool}); e_1 \mapsto \sigma \triangleright I; K; e_1
$$

**red-assert-t**

$$
\sigma \triangleright I; K; x = e_1; e_2 \mapsto \sigma \triangleright I; (x.e_2) :: K; e_1
$$

**red-let**

$$
\sigma \triangleright C \mapsto C'
$$

**Internal reduction**

$$
\sigma \triangleright T_i \mapsto \sigma' \triangleright T'_i
\sigma \triangleright T_1, \ldots, T_i, \ldots, T_n \mapsto \sigma' \triangleright T_1, \ldots, T'_i, \ldots, T_n
$$

**red-config**

$$
\sigma \triangleright T_1, \ldots, (I_j; (x.e_1) :: K_1; b_1 : \text{send}(t)), \ldots, (I_k; (x.e_2) :: K_2; b_2 : \text{recv}(t)), \ldots, T_n
$$

$$
\langle(I_j, b_1), (I_k, b_2)\rangle \sigma' \triangleright T_1, \ldots, (I_j; K_1; e_1[0/x]), \ldots, (I_k; K_2; e_2[t/x]), \ldots, T_n
$$

**red-act**

$$
6.2 \text{ Case study: Compromising notaries attack}
$$

We model an instance of our running example based on passwords in order to demonstrate our actual cause definition. As explained in Section 2, we consider a protocol session where User1, Server1 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) and a second user (User2) who follow their norms for account creation. These threads do not interact with threads $\{\text{User1, Server1, Notary1, Notary2, Notary3, Adversary}\}$. The protocol has been described in detail below.
**Protocol Description** We consider our example protocol with nine threads named \{Server1, User1, Adversary, Notary1, Notary2, Notary3, Notary4, Server2, User2\}. The norms for all these threads, except Adversary are shown in Figure 3 and 4. The Adversary’s norm is empty, because in an ideal world, the Adversary should not participate. In this case study, we have two servers (Server1, Server2) running the protocol with two different users (User1, User2) and each server allocates account access separately.

The norms in Figures 3, 4 assume that User1’s and User2’s accounts (called acct1 and acct2 in Server1’s and Server2’s norm respectively) have been created already. User1’s password, pwd1 is associated with User1’s user id uid1. Similarly User2’s password pwd2 is associated with its user id uid2. User1 generated pwd1 associated with acct1 and User2 created pwd2 associated with acct2. This association (in hashed form) is stored in Server1’s local state at pointer mem1 (and at mem2 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 mem1. To grant access, Server1 adds an entry into a private access matrix, called P1. (A separate server thread, not shown here, allows User1 to access its resource if this entry exists in P1.)

The norm for User1 is to send the password pwd1 to Server1, encrypted under Server1’s public key and not share the password with any other agent. 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 User2 is the same as that for User1 except that it interacts with Server2. Note that User2 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 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 User2 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.)

**Preliminaries**

**Notation** The norms in Figures 3, 4 use several primitive functions. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For a given (public or private) key k, Inv(k) denotes the corresponding private or public key for k. For readability, we include the intended recipient i and expected sender j of a message as the first argument of send(i, m) and recv(j) expressions. As explained earlier, i and j are ignored during execution and the adversary, if present, may capture or inject messages. \(P_1(u)\) 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.
**Action Predicates and terms**

- **Send**(i, j, m) @ u holds if thread i sends message m to thread j at time u.
- **Recv**(i, j, m) @ u holds if thread i receives message m from thread j at time u.
- **Enc**(k, m) and **Dec**(k', m) denote encryption and decryption of message m with key k and k' respectively.
- The action **new** generates a new value, the predicate **New**(i, n) @ u holds when thread i generates nonce n at time u.
- The action **sig**(l, v) signs v with key l and returns **Sig**(l, v) while **verify**(l', v) verifies v of the form **Sig**(l, v') and returns **true** if the signature is valid.
- The action **hash**(m, m1) generates the hash of terms (m, m1) and returns **Hash**(m, m1).
- The action **read**(r) reads a location r and the corresponding predicate **Read**(i, r) @ u holds if thread i reads resource r at time u. The action **write**(r, d) @ u writes value d to a resource r and the corresponding predicate **Write**(i, r, d) @ u holds if thread i writes value d at location r at time u. For instance in our example, **Server**1 writes to **mem**1.
- **Mem**1(t) @ u holds if **mem**1 contains t at time u. **Mem**2(t) @ u holds if **mem**2 contains t at time u.

**Assumptions:**

(A1) \( \text{HonestThread} \left( \text{Server}_1, \mathcal{N}(\text{Server}_1) \right) \)

We are interested in security guarantees about honest 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 norms.

(A2) \( \text{HonestThread} \left( \text{User}_1, \mathcal{N}(\text{User}_1) \right) \)

(A3) \( \text{HonestThread} \left( \text{Adversary}, \mathcal{A}(\text{Adversary}) \right) \)

(A4) \( \text{HonestThread} \left( \text{Notary}_1, \mathcal{A}(\text{Notary}_1) \right) \)

(A5) \( \text{HonestThread} \left( \text{Notary}_2, \mathcal{A}(\text{Notary}_2) \right) \)

(A6) \( \text{HonestThread} \left( \text{Notary}_3, \mathcal{A}(\text{Notary}_3) \right) \)

(A7) \( \text{HonestThread} \left( \text{Notary}_4, \mathcal{N}(\text{Notary}_4) \right) \)

(A8) \( \text{HonestThread} \left( \text{Server}_2, \mathcal{N}(\text{Server}_2) \right) \)
A principal following the protocol never shares its keys with any other entity. We also assume that the encryption scheme is 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 N(Server1).

(Start1) 
\[ \text{Start}(i) @ -\infty \]

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

**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 \( k = \text{User}_1 \). This can be formalized by the following logical formula, \( \neg \varphi_U \):

\[
\forall u, k. (acct_1, k) \in P_1(u) \supset (k = \text{User}_1)
\]

Here, \( P_1(u) \) is the state of the access control matrix \( P_1 \) for Server1 at time \( u \). (We use this logical formalization of the property in establishing the actual causes using our definition. Specifically, we use a program logic to establish the sufficiency and necessity conditions.)

**Attack** As an illustration, we model the “Compromising Notaries (Man in the Middle)” violation of Section 2. In this attack scenario, User1 and Server1 execute norms. 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 Adversary. User2, Server2 and Notary4 execute their norms in order to access the account acct2 as well.

In the property-violating trace, User1, Server1, User2, Server2 and Notary4 execute their norms and the expressions executed by Adversary and the three deviant notaries are shown in Figure 5. Thus, the initial configuration has the programs: \( \{N(\text{User}_1), N(\text{Server}_1), A(\text{Adversary}), A(\text{Notary}_1), A(\text{Notary}_2), A(\text{Notary}_3), N(\text{User}_2), N(\text{Server}_2), N(\text{Notary}_4)\} \).

For this attack scenario, the concrete trace we consider is any arbitrary interleaving of the logs for \( X = \{\text{Adversary}, \text{User}_1, \text{Server}_1, \text{Notary}_1, \text{Notary}_2, \text{Notary}_3\} \) and \( Y = \{\text{Server}_2, \text{User}_2, \text{Notary}_4\} \) shown in Figure 6. Any such interleaved log is denoted \( l \) in the sequel.

At the end of \( l \), \( (acct_1, \text{Adversary}) \) occurs in the access control matrix \( P_1 \), but Adversary does not own acct1. Hence, this log corresponds to a violation of our security property. Importantly, even though only Adversary and three notaries
Fig. 3. Norms for Server1, User1, notaries. Adversary’s norm is the trivial empty program.

deveiate from their norms, this violation cannot happen without participation from User1 and Server1. Moreover, if any two of the three notaries had deviated from their norm, the violation would have still happened. Consequently, we may expect three independent causes in this example: \{Adversary, User1, Server1, Notary1, Notary2\}, \{Adversary, User1, Server1, Notary1, Notary3\}, and \{Adversary, User1, Server1, Notary2, Notary3\}. The following theorem states that our definitions determine exactly these three independent causes.

**Theorem 2.** Let \(I = \{User1, Server1, Adversary, Notary1, Notary2, Notary3, Notary4, Server2, User2\}\), \(l\) be a log defined above, \(t\) be a trace with \(log(t) = l\) and \(A, N\) and \(\Sigma\) be as described above. Let \(V = \{I, A, \Sigma, N, \varphi_V, t\}\). Then, Definition 11 determines three possible values for the actual cause \(X_d\) of violation: \{Adversary, User1, Server1, Notary1, Notary2\}, \{Adversary, User1, Server1, Notary1, Notary3\}, and \{Adversary, User1, Server1, Notary2, Notary3\}.
**Norm $N'(Server2)$:**

1: $(uid2, n1) = recv(j)$; //access req from thread $j$
2: $n2$ = new;
3: $send(j, Sig(pub_key_Server2, (pub_key_Server2, n2, n1)))$;
4: $s1 = recv(j)$; //encrypted $uid2, pwd2$ from $j$, alongwith its thread id $J$
5: $(n2, uid2, pwd2, J) = Dec(pub_key_Server2, s1)$;
6: $t = Hash(uid2, pwd2)$;
7: $assert($mem$_2 = t$) //compare hash with stored hash value for same uid
8: $insert(P_2, (acct2, J))$;

**Fig. 4.** Norms for Server2, User2

**Actual program $A(Adversary)$**

1: $(uid1, n1) = recv(j)$; //intercept req from User1
2: $n2$ = new;
3: $send(User1, pub_key_Adversary)$; //send key to User1
4: $s = recv(User1)$; //pwd from User
5: $n2, uid1, pwd1, User1 = Dec(pub_key_Adversary, s)$; //decrypt pwd;
6: $n3$ = new;
7: $send(Server1, (uid1, n3))$; //access request to Server
8: $pub_key, n4, n3 = recv(Server1)$;
9: $t = Enc(pub_key, (n4, uid1, pwd1, Adversary))$; //encrypt pwd
10: $send(Server1, t)$; //pwd to Server1

**Actual programs $A(Notary1), A(Notary2), A(Notary3)$**

// $o$ denotes Notary1, Notary2 or Notary3
1: $(pub_key_Adversary, n1) = recv(j)$;
2: $send(j, Sig(pub_key_o, (pub_key_Adversary, Server1, n1)))$; //signed certificate to $j$;

**Fig. 5.** Deviants for Adversary and Notary1, Notary2, Notary3
or X and X sufficiency. (Since obtain a violation, so sufficiency cannot be satisfied for these two values of X already established in necessity that starting from all nine norms, we cannot following four values of X: \( X_1 = I, X_2 = \emptyset, X_3 = \{\text{Notary4, Server2, User2}\} \) and \( X_4 = \{\text{Adversary, User1, Server1, Notary1, Notary2, Notary3}\} \). When \( X = X_2 \) or \( X = X_3 \), \( \text{norm}(V, X, Y) \) contains only norms for all nine threads. We have already established in necessity that starting from all nine norms, we cannot obtain a violation, so sufficiency cannot be satisfied for these two values of X. That leaves only \( X = X_1 \) and \( X = X_4 \). We now show that \( X = X_4 \) satisfies sufficiency. (Since \( X_4 \subseteq X_1 \), minimality then implies that the X determined by Definition \text{4} must be \( X_4 \).) Following the statement of sufficiency, let \( T \) be the set

| Synchronization projection \( f(I(V,X,Y)) \) |
|---|
| \( \langle\text{User1}, 2\rangle, \langle\text{Adversary}, 1\rangle, \) |
| \( \langle\text{Adversary}, 3\rangle, \langle\text{User1}, 3\rangle, \) |
| \( \langle\text{User1}, 5\rangle, \langle\text{Notary1}, 1\rangle, \) |
| \( \langle\text{User1}, 6\rangle, \langle\text{Notary2}, 1\rangle, \) |
| \( \langle\text{User1}, 7\rangle, \langle\text{Notary3}, 1\rangle, \) |
| \( \langle\text{Notary1}, 3\rangle, \langle\text{User1}, 8\rangle, \) |
| \( \langle\text{Notary2}, 3\rangle, \langle\text{User1}, 9\rangle, \) |
| \( \langle\text{Notary3}, 3\rangle, \langle\text{User1}, 10\rangle, \) |
| \( \langle\text{User1}, 13\rangle, \langle\text{Adversary}, 4\rangle, \) |
| \( \langle\text{Adversary}, 7\rangle, \langle\text{Server1}, 1\rangle, \) |
| \( \langle\text{Server1}, 3\rangle, \langle\text{Adversary}, 8\rangle, \) |
| \( \langle\text{Adversary}, 10\rangle, \langle\text{Server1}, 4\rangle \) |

| Synchronization projection \( f(I(V,X,Y)) \) |
|---|
| \( \langle\text{User2}, 1\rangle, \langle\text{Server2}, 1\rangle, \) |
| \( \langle\text{Server2}, 3\rangle, \langle\text{User2}, 3\rangle, \) |
| \( \langle\text{User2}, 5\rangle, \langle\text{Notary4}, 1\rangle, \) |
| \( \langle\text{Notary4}, 3\rangle, \langle\text{User2}, 6\rangle, \) |
| \( \langle\text{User2}, 9\rangle, \langle\text{Server2}, 4\rangle \) |

Fig. 6. Synchronization projections

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that \( V \) is a violation structure and that our Phase 1 and Phase 2 definitions yield exactly the three values for \( X_d \) mentioned in the theorem. To check that \( V \) is a violation structure, we must verify the occurrence and necessity conditions from Definition \[8\]. For occurrence, we must show that \( t \in \varphi_V \). This is clear because at the end of the trace, \( (acct_1, \text{Adversary}) \in P_1 \), but \( \text{Adversary} \neq \text{User1} \). For necessity, we show that any trace starting from \( \langle I, N, \Sigma \rangle \) cannot be in \( \varphi_V \), i.e., any such trace has the invariant \[2\] [Security property]. To prove this, we can use any sound program logic. In particular, we use the logic of Garg \textit{et al.} [8]. Our proof, in fact, shows the stronger property that \[2\] is an invariant even when the nine norms execute concurrently with any number of standard Dolev-Yao adversaries (we assume that encryption is non-malleable). This is just a standard proof of protocol correctness with a program logic.

\textbf{Phase 1}: The closure condition of Phase 1 (Definition \[9\]) can be satisfied by the following four values of \( X: X_1 = I, X_2 = \emptyset, X_3 = \{\text{Notary4, Server2, User2}\} \) and \( X_4 = \{\text{Adversary, User1, Server1, Notary1, Notary2, Notary3}\} \). When \( X = X_2 \) or \( X = X_3 \), \( \text{norm}(V, X, Y) \) contains only norms for all nine threads. We have already established in necessity that starting from all nine norms, we cannot obtain a violation, so sufficiency cannot be satisfied for these two values of \( X \). That leaves only \( X = X_1 \) and \( X = X_4 \). We now show that \( X = X_4 \) satisfies sufficiency. (Since \( X_4 \subseteq X_1 \), minimality then implies that the \( X \) determined by Definition \[4\] must be \( X_4 \).) Following the statement of sufficiency, let \( T \) be the set
of traces starting from \( \text{norm}(V, X_d, I \setminus X_d) \) that are log-consistent with \( l \) relative to \( X_d \). Note that \( \text{norm}(V, X_d, I \setminus X_d) \) is exactly the same as the original initial configuration since threads in \( I \setminus X_d = \{\text{Notary}4, \text{Server}2, \text{User}2\} \) were already executing norms. This implies that \( t \in T \), so \( T \) is non-empty. If we pick any other \( t' \in T \), then because the log of \( t' \) is consistent with \( l \) relative to \( X_d \), that log must be exactly \( l \setminus \{\text{Adversary}, \text{User}1, \text{Server}1, \text{Notary}1, \text{Notary}2, \text{Notary}3\} \) from Figure [6]. This trivially implies that \( t' \in \varphi_V \). Hence, Phase 1 determines \( X = X_d = \{\text{Adversary}, \text{User}1, \text{Server}1, \text{Notary}1, \text{Notary}2, \text{Notary}3\} \).

**Phase 2:** Phase 2 (Definition 11) determines three independent actual causes for \( X_d \): \{Adversary, User1, Server1, Notary1, Notary2\}, \{Adversary, User1, Server1, Notary1, Notary3\}, and \{Adversary, User1, Server1, Notary2, Notary3\}. These are symmetric, so we only explain why \( X_d = \{\text{Adversary}, \text{User}1, \text{Server}1, \text{Notary}1, \text{Notary}2\} \) satisfies Definition 11. We show that (a) This set \( X_d \) satisfies sufficiency', and (b) No proper subset of \( X_d \) satisfies sufficiency' (minimality').

We start with (a). By definition, \( X_p = X \setminus X_d = \{\text{Notary}3\} \). Fix any dummyifying function \( f \). We must show two things. First, there is a trace from \( \text{dummyify}(V, X_d, X_p, f) \), whose log is consistent with \( l \) relative to \( X_d \). Second, any such trace is in \( \varphi_V \). The first statement is easily established by mimicking the execution in \( l \) starting from \( \text{dummyify}(V, X_d, X_p, f) \). The only potential issue is that on line 7, \( \mathcal{A}(\text{User}1) \) (which equals \( \mathcal{N}(\text{User}1) \) from Figure 3) synchronizes with \( \text{Notary}3 \) according to \( l \), but \( \text{Notary}3 \) does not exist in \( \text{dummyify}(V, X_d, X_p, f) \). However, in \( \text{dummyify}(V, X_d, X_p, f) \), the \text{recv()} on line 10 in \( \mathcal{A}(\text{User}1) \) is replaced with a dummy value, so the execution from \( \text{dummyify}(V, X_d, X_p, f) \) progresses. Subsequently, the majority check on line 11 succeeds as in \( l \), because two of three notaries (\( \text{Notary}1 \) and \( \text{Notary}2 \)) still attest the \text{Adversary}'s key.

Next we prove that every trace starting from \( \text{dummyify}(V, X_d, X_p, f) \), whose log is consistent with \( l \) relative to \( X_d \), is in \( \varphi_V \). Fix a trace \( t' \) with log \( t' \). Assume \( l' \) consistent with \( l \) relative to \( X_d \). We show \( t' \in \varphi_V \) as follows:

1. Since \( t' \) is consistent with \( l \) relative to \( X_d \), \text{Server}1 must execute line 8 of its program \( \mathcal{N}(\text{Server}1) \) in \( t' \). After this line, the access control matrix \( P_1 \) contains \( \text{acct}1, J \) for some \( J \).
2. When \( \mathcal{N}(\text{Server}1) \) writes \((x, J)\) to \( P_1 \) at line 8, then \( J \) is the third component of a tuple obtained by decrypting a message received on line 4.
3. Since \( t' \) is consistent with \( l \) relative to \( X_d \), and on \( l \) (\text{Server}1, 4) synchronizes with \( \{\text{Adversary}, 10\} \), \( J \) must be the third component of an encrypted message sent on line 10 of \( \mathcal{A}(\text{Adversary}) \).
4. The third component of the message sent on line 10 by \text{Adversary} is exactly the term "\text{Adversary}". (This is easy to see, as the term "\text{Adversary}" is hardcoded on line 9.) Hence, \( J = \text{Adversary} \).
5. This immediately implies that \( t' \in \varphi_V \) since \((\text{acct}1, \text{Adversary}) \in P_1 \), but \( \text{Adversary} \neq \text{User}1 \).

Last, we prove (b) — that no proper subset \( X'_d \) of \( X_d \) satisfies sufficiency'. Pick any \( X'_d \subseteq X_d \) such that \( X'_d \) satisfies sufficiency'. We show that \( X'_d = X_d \). Let \( X'_p = X \setminus X'_d \). Observe that if \( \text{Server}1 \notin X'_d \), then \( \text{Server}1 \) is not in
dummify\((V, X'_d, X'_p, f)\) and, hence, on any counterfactual trace cannot write to \(P\), thus precluding a violation. Therefore, Server1 \(\in X'_d\). By sufficiency’, for any \(f\), the log \(l'\) of at least one trace \(t'\) of dummify\((V, X'_d, X'_p, f)\) must be consistent with \(l\) relative to \(X'_d\). This means that in \(t'\), the assertion on line 7 of \(N\)(Server1) must succeed and, hence, on line 5, the correct password \(pwd_1\) must be received by Server1, independent of \(f\). This immediately implies that Adversary (which sends that password on \(l\)) must be in \(X'_d\), else some dummified executions of Server1 will have the wrong password and the assertion on line 7 will fail.

Extending this logic further, we now observe that because Adversary forwards a password received from User1 (line 4 of \(A\)(Adversary)) to Server1, User1 \(\in X'_d\) (otherwise, some dummifications of line 4 of \(A\)(Adversary) will result in the wrong password being sent to Server1, a contradiction). Since User1 \(\in X'_d\) and \(l'\) must be consistent with \(l\) relative to \(X'_d\), the majority check on line 11 of \(N\)(User1) must also succeed. This means that at least two of \{Notary1, Notary2, Notary3\} must be in \(X'_d\), else the dummification of lines 8 – 10 of \(N\)(User1) will cause the assertion check to fail for some \(f\). Since \(X'_d \subseteq X_d\), these two threads must be \{Notary1, Notary2\}. At this point we have established that each of \{Adversary, User1, Server1, Notary1, Notary2\} is in \(X'_d\). Hence, \(X'_d = X_d\).

Violation Structure (Necessity) : If all programs execute norms, there will be no violation on any resulting trace due to the correctness of the protocol as described next:

1. We prove that if access permission \((acct_1, k)\) is added in \(P_1\) for principal \(k\), then \(k = User1\). Initially only User1 and Server1 know the password \(pwd_1\) stored in Server1’s memory \(mem_1\).
2. Consider the traces where \((acct_1, k) \notin P_1 @ u\) for some \(k\). It is an (easy to prove) invariant that Server1 only adds access permission for \(acct_1\) if a principal sends an access request with the correct password \(pwd_1\) stored in \(mem_1\). Therefore, Server1 must have received the correct password for \(pwd_1\) from some thread \(j\).
3. We prove an invariant of \(N\)(User1) that if it sends out a password encrypted under a public key, then the public key was verified by the notaries to be Server1’s public key. Similarly, we prove an invariant of the norms of the notaries that they only certify correct keys. Consequently, User1 only sends the password encrypted under Server1’s public key.
4. It is an invariant of \(N\)(Server1) that it never sends any password.
5. Therefore, only User1 and Server1 ever see the password \(pwd_1\).
6. Hence, the password received by Server1 in step (2) must have been sent by \(j = User1\).
7. It is an invariant of \(N\)(User1) that if it sends a request to add an access permission (line 8), it does so for itself. Combining with (2), we deduce that \(k = User1\).

---

4 The proof technique for Necessity follows the proof of secrecy for Kerberos protocol by Garg et al. [8].
Expanding the steps: We adapt the definitions from the proof of secrecy for Kerberos by Garg et al [9] for a framework based on asymmetric encryption.

Some of these definitions are given in Figure 7. For more details, we refer the reader to [9]. Here $K$ denotes a set of keys. $\text{OwnedIn}(i, S)$ means that thread $i$ in set $S$, $\text{OrigRes}(s, S)$ means that the thread which created the term $s$ lies in set $S$. $\text{KeyRes}(K, S)$ means that the inverse key corresponding to each key in set $K$ is known only to threads in $S$. $\text{KORes}(s, K, u)$ combines the previous two predicates, $\text{SendsSafeMsg}(i, s, K)$ means that if thread $i$ sends $s$ in a message, then all occurrences of $s$ in that message are protected by keys in $\text{Inv}(K)$ i.e. the private keys corresponding to the public keys in $K$. $\text{SafeNet}(s, K, u)$ means that prior to time $u$, every thread protects $s$ in all messages it sends using keys in $\text{Inv}(K)$.

Figure 7 also lists two axioms (POS) and (NET) from prior work. We introduce an additional axiom called (NM). This axiom (assuming non-malleability) states that if only principals in set $P$ produce an additional axiom called (NM). This axiom (assuming non-malleability) states that if only principals in set $P$ had access to the corresponding private keys for set $K$ and have knowledge of terms in $s$, then if a message $m$ containing secret $s$ is encrypted under a key $k \in K$ and sent at time $u$, then at a prior time $u'$ some principal in $P$ must have sent the message on the network.

We prove that if access permission ($\text{acct}_1, k$) is added in $P_1$ for principal $k$, then $k = \text{User}_1$.

1. Consider the traces where ($\text{acct}_1, k$) $\in P_1$ @ $u$ for some $k$. We will first prove that $\text{Server}_1$ only adds access permission for $\text{acct}_1$ if a principal sends an access request with the correct password $pwd_1$ stored in $\text{mem}_1$. Therefore, $\text{Server}_1$ must have received the correct password for $\text{acct}_1$ from some thread $j$ at time $u_1 < u$.

Invariant of $\text{Server}_1$:

$$\{N(\text{Server}_1)\} \langle u_b, u_e, i \rangle \forall u, k. ((u_b < u \leq u_e) \land \text{Insert}(i, (\text{acct}_1, k)) \circ u) \supset (\exists j, k, n_1, pwd_1, u_1, (u_1 < u) \land \text{Mem}_1(\text{Hash}(uid_1, pwd_1)) \circ u \land$$

$$\text{Recv}(\text{Server}_1, j, (\text{Enc}(\text{pub_key}_\text{Server}_1, (n_1, uid_1, pwd_1, k))) \circ u_1)$$

which we abbreviate as:

$$\{N(\text{Server}_1)\} \langle u_b, u_e, i \rangle \forall u, k. ((u_b < u \leq u_e) \land \text{Insert}(i, (\text{acct}_1, k)) \circ u) \supset \varphi$$

Using (HONTH) in conjunction with [11] assuming (Start1), we get:

$$\forall u'. (u' > -\infty) \supset \forall u, k. ((u_b < u \leq u_e) \land \text{Insert}(\text{Server}_1, (\text{acct}_1, k)) \circ u) \supset \varphi$$

Choosing $u_e = \infty$, we get:

$$\forall u, k. \text{Insert}(\text{Server}_1, (\text{acct}_1, k)) \circ u \supset \varphi$$

We have previously assumed that $\text{Server}_1$ inserts permission for principal $k$, therefore the consequent of the above implication holds. This implies that at some time $u_1 < u$, $\text{Server}_1$ received the correct password $pwd_1$ for $\text{acct}_1$ from some thread $j$. In the sequel we use $pwd_1$ to denote the password stored in $\text{mem}_1$. 
Definitions

OwnedIn(i, S) = i ∈ S
OrigRes(s, S) = ∀i, u. New(i, s) @ u ⊃ OwnedIn(i, S)
KeyRes(K, S) = ∀i, k. (Has(i, Inv(k)) ∧ k ∈ K) ⊃ OwnedIn(i, S)
KORes(s, K, S) = OrigRes(s, S) ∧ ∀u. KeyRes(K, S) @ u
SendsSafeMsg(i, s, K) = Send(i, v) ⊃ SafeMsg(v, s, K)
SafeNet(s, K, u) = ∀i, u'. (u' ≤ u) ⊃ SendsSafeMsg(i, s, K) @ u'
HasOnly(S, s) = ∀i. Has(i, s) ⊃ OwnedIn(i, S)

Additional Axioms

(NET) (KORes(s, K, S) ∧ SafeNet(s, K, u1) ∧ ~SafeNet(s, K, u2) ∧ (u1 < u2)) ⊃
  (∃i, u3. (u1 < u3 ≤ u2) ∧ OwnedIn(i, S) ∧ ~SendsSafeMsg(i, s, K) @ u3 ∧
  ∀u4 ∈ (u1, u3). SafeNet(s, K, u4)

(POS) (SafeNet(s, K, u) ∧ Has(i, s) @ u) ⊃
  (∃u'. (u' < u) ∧ New(i, s) @ u') ∨
  (∃k. (k ∈ K) ∧ Has(i, Inv(k)) @ u)

(NM) (HasOnly(In(K), P) @ u ∧
  HasOnly(s, P) @ u ∧ ∃i1. Send(i1, m') @ u ∧
  m' = Enc(k, m) ∧ (k ∈ K) ∧ Contains(m, s)) ⊃
  (∃u', j1. (u' < u) ∧ j ∈ P ∧ (Send(j1, m) @ u'))

Fig. 7. Additional definitions and axioms (9)

2. Initially only User1 and Server1 know the password pwd1 stored in Server1’s memory mem1:

HasOnly({Server1, User1}, pwd1) @ −∞

We also assume that ∀u. HasOnly(Server1, pubKey_Server1) @ u.

3. Next we prove that only User1 and Server1 ever see the password pwd1. First, we prove that SendsSafeMsg(User1, pwd1, pubKey_Server1) @ u for every u. We prove an invariant of N(User1) that if it sends out a password encrypted under a public key, then the public key was verified by the notaries to be Server1’s public key. Note that User1 only sends out the password in line 13 of the code N(User1).
Invariant of User1:

\[
\{N(\text{User}1)\}\langle u_b, u_e, i \rangle \forall u, j_1, n_1 . ((u_b < u < u_e) \land \\
\text{Send}(\text{User}1, j_1, (\text{Enc}(\text{pub}_j, j_1, \{n_1, \text{uid}_1, \text{pwd}_1, \text{User}1\})) \@ u) \supset \\
(\exists n_2, u_1 . (u_1 < u) \land \\
\text{Recv}(\text{User}1, \text{Notary}1, \text{Sig}(\text{pub}_j, \text{Notary}1, (\text{pub}_j, \text{Server}1, n_2)) \@ u_1))
\]

We abbreviate the consequent of the above implication as \(\varphi_2\).

Using (HONTH) in conjunction with 3 above, assuming (Start1), we get:

\[
\forall u'. (u' > -\infty) \supset \forall u, j_1, n_1, \text{pwd}_1 . \\
((u_b < u < u_e) \land \text{Send}(\text{User}1, j_1, (\text{Enc}(\text{pub}_j, j_1, \{n_1, \text{uid}_1, \text{pwd}_1, \text{User}1\})) @ u) \supset \varphi_2
\]

Choosing \(u_e = \infty\), we get

\[
\forall u, j_1, n_1, \text{pwd}_1 . \text{Send}(\text{User}1, j_1, (\text{Enc}(\text{pub}_j, j_1, \{n_1, \text{uid}_1, \text{pwd}_1, \text{User}1\})) \@ u \supset \varphi_2
\]

4. Similarly, we prove an invariant of the norms of the notaries that they only certify correct keys. We assume that the notary repository is correct. By analyzing the notaries’ threads \(N(\text{Notary}1), N(\text{Notary}2), N(\text{Notary}3), N(\text{Notary}4)\), we prove that if the notaries sign a key stating that it belongs to Server1 (or Server2), the key must belong to Server1 (or Server2 respectively). We show the invariant for one of the notaries, Notary1 which we abbreviate as:

\[
\{N(\text{Notary}1)\}\langle u_b, u_e, i \rangle \forall u, j_1, k, \text{pwd}_1 . \\
(u_b < u < u_e) \land \text{Send}(\text{Notary}1, j_1, (\text{Sig}(\text{pub}_k, \text{Notary}1, (\text{pub}_k, \text{Server}1, n_1))) \@ u \supset \\
\text{pub}_k = \text{pub}_\text{Server}1
\]

Consequently, User1 only sends the password encrypted under Server1’s public key. This implies that User1 only sends safe messages containing \(\text{pwd}_1\) and encrypted under Server1’s public key.

5. Next, it is trivial to prove that \(\text{SendsSafeMsg}(\text{Server}1, \text{pwd}_1, \text{pub}_k, \text{Server}1) \@ u\) for every \(u\). This is because Server1 never sends out the password in any message.

6. Next we apply rely-guarantee reasoning similar to the secrecy proof for Kerberos by Garg et al [9] in order to show \(\forall u. \varphi(u)\) where

\[
\varphi(u) = \forall u. \text{SafeNet}(\text{pwd}_1, \text{pub}_k, \text{Server}1, u)
\]

We instantiate the framework of rely-guarantee by choosing:

\[
i(i) = (i = \text{Server}1) \land (i = \text{User}1)
\]

\[
\psi(u, i) = \text{SendsSafeMsg}(i, \text{pwd}_1, \text{pub}_k, \text{Server}1)
\]

\[
\varphi(u) = \text{SafeNet}(\text{pwd}_1, \text{pub}_k, \text{Server}1, u)
\]

In order to apply the method of rely-guarantee, we must show that the following hold for \(\varphi, \iota, \psi\) as defined above:
To prove (3), we instantiate the axiom (NET) by choosing $s = pubd_1$, $K = \{\text{pub key, Server}\}$, and $S_0 = \{\text{Server1, User1}\}$ to obtain:

\[
\begin{align*}
&\text{KORes}(pwd_1, \text{pub key, Server1}, S_0) \land \text{SafeNet}(pwd_1, \text{pub key, Server1}, u_1) \land \\
&\quad \neg\text{SafeNet}(pwd_1, \text{pub key, Server1}, u_2) \land (u_1 < u_2) \supset \\
&\quad \exists i, u_3. (u_1 < u_3 \leq u_2) \land \text{OwnedIn}(i, S_0) \land \\
&\quad \neg\text{SendsSafeMsg}(i, S_0, \text{pub key, Server1}) @ u_3 \land \\
&\quad \forall u_4 \in (u_1, u_3). \text{SafeNet}(pwd_1, \text{pub key, Server1}, u_4)
\end{align*}
\]

We show that KORes($pwd_1, \text{pub key, Server1}, \{\text{User1, Server1}\}$). Expanding the definition of KORes, we need to show that $pwd_1$ was generated by either User1 or Server1 (which is true by assumption since User1 generated $pwd_1$) and that $\text{pub key, Server1}$ is known only to $\{\text{User1, Server1}\}$ (assumption). Therefore, KORes($pwd_1, \text{pub key, Server1}, S_0$) holds and we eliminate that condition from the above formula to obtain:

\[
\begin{align*}
&(\text{SafeNet}(pwd_1, \text{pub key, Server1}, u_1) \land \neg\text{SafeNet}(pwd_1, \text{pub key, Server1}, u_2) \land \\
&\quad (u_1 < u_2)) \supset \\
&\quad \exists i, u_3. (u_1 < u_3 \leq u_2) \land \text{OwnedIn}(i, S_0) \land \\
&\quad \neg\text{SendsSafeMsg}(i, S_0, \text{pub key, Server1}) @ u_3 \land \\
&\quad \forall u_4 \in (u_1, u_3). \text{SafeNet}(pwd_1, \text{pub key, Server1}, u_4)
\end{align*}
\]

This proves the statement of (3) above. Hence, we deduce that $\forall u. \varphi(u)$, i.e.

\[
\forall u. \text{SafeNet}(pwd_1, \text{pub key, Server1}, u)
\]

(3)

Next, we fix any time parameter $u'_0$, and try to show that $\text{HasOnly}(S_0, pwd_1) @ u'_0$. Following the definition of $\text{HasOnly}$ assume that for some $i$, $\text{Has}(i, pwd_1) @ u'_0$. It suffices to show that $\text{OwnedIn}(i, S_0)$. From (3) above, the assumption $\text{Has}(i, pwd_1) @ u'_0$, and axiom (POS), we obtain:

\[
(\exists u'. (u' < u'_0) \land \text{New}(i, pwd_1) @ u') \lor \\
(\exists k. (k \in \{'\text{pub key, Server1}'}\) \land \text{Has}(i, \text{Inv}(k)) @ u'_0)
\]

We case analyze these two disjuncts. If $\exists u'. (u' < u'_0) \land \text{New}(i, pwd_1) @ u'$, then we obtain $i = \text{User1}$. Since $S_0 = \{\text{User1, Server1}\}$, $\text{OwnedIn}(i, S_0)$ follows from definition of $\text{OwnedIn}$. If $\exists k. (k \in \{'\text{pub key, Server1}\}') \land \text{Has}(i, \text{Inv}(k)) @ u'_0$, then from the assumptions that $\text{User1}$ generated $pwd_1$ and $\text{Server1}$’s public key is only known to $\text{Server1}$, i.e., KORes($pwd_1, \text{pub key, Server1}, S_0$), we immediately obtain $\text{OwnedIn}(i, S_0)$. Since, in both case analyses we obtain $\text{OwnedIn}(i, S_0)$, it follows that $\text{HasOnly}(S_0, pwd_1) @ u'_0$ for any $u'_0$. Since $u'_0$ is a parameter, this implies $\forall u'. \text{HasOnly}(S_0, pwd_1) @ u'$, which is the property we wanted to prove, i.e.

\[
\forall u. \text{HasOnly}(\{'\text{Server1, User1}\'}, pwd_1) @ u
\]
7. From 6. above, $pwd_1$ is only known to $S_0$. By assumption,

$$\forall u. \text{HasOnly}([\text{Server}_1], \text{priv_key}_{\text{Server}_1}) @ u$$

Also initially we showed that a message was sent to $\text{Server}_1$ which contained the password encrypted under $\text{Server}_1$’s public key. Instantiating the antecedent in (NM) with $P = S_0$, $s = pwd_1$, $K = \text{pub_key}_{\text{Server}_1}$, we can infer that either $\text{User}_1$ or $\text{Server}_1$ sent the initial message containing the password encrypted under $\text{Server}_1$’s key.

8. It is an invariant of $\mathcal{N}(\text{Server}_1)$ that it never sends any password.

9. It is an invariant of $\mathcal{N}(\text{User}_1)$ that if it sends a request to add an access permission (line 8), it does so for itself. Invariant of $\text{User}_1$:

$$\{\mathcal{N}(\text{User}_1)\} \langle u_b, u_e, i \rangle \forall u, j, n. \left((u_b < u \leq u_e) \land 
\text{Send}(\text{User}_1, j, (\text{Enc}(% \text{pub_key}_j, (n1, \text{uid}_1, pwd_1, k)))) @ u \supset
(k = \text{User}_1)\right)$$

Combining with (2), we deduce that $k = \text{User}_1$. Therefore, we have prove that if access permission $(acct_1, k)$ is added in $P_1$ for principal $k$, then $k = \text{User}_1$. 