Securing Causal Relationships in Distributed Systems

MICHAEL REITER AND LI GONG*

AT&T Bell Laboratories, Murray Hill, New Jersey, USA
**SRI International, Menlo Park, California, USA*
Email: reiter@research.att.com,gong@csl.sri.com

In a distributed system, it is often important to detect the causal relationships between events, where event e_1 is causally before event e_2 if e_1 happened before e_2 and could possibly have affected the occurrence of e_2. In this paper we argue that it can be essential to security that a process determine, in the face of malicious attack, how two events are causally related. We formulate attacks on causality detection in terms of causal denial and forgery, formalize possible security goals with respect to causality, and present simple algorithms to attain these goals in some situations.

Received October 18 1994, revised October 19 1995

1. INTRODUCTION

In a distributed system, it is often important to detect the causal relationships between events, where event e_1 is causally before event e_2 if e_1 happened before e_2 and could possibly have affected the occurrence of e_2 [1]. Causality has been recognized as fundamental to distributed computing and forms the basis for event orderings in many distributed systems and distributed service implementations (e.g. [2–6]). For instance, several systems implement communication primitives that deliver messages in an order consistent with the causal relationships among the messages (i.e. among the events in which the messages were sent). This causal order can be seen as an extension of FIFO order to a setting with multiple senders and receivers, and is especially useful in systems that exploit asynchronous communication for performance [7].

Here we argue that it can be essential to security that a process determine, in the face of malicious attack, how two events are causally related. The view of causality that we take is very different from that taken by previous treatments of causality in the security literature. Previous studies of causality and security have occurred in the context of multi-level information flow, where one goal is, informally, to prevent events at higher-level objects from causally preceding events at, and thus carrying information to, lower-level objects. That is, in previous works, causal relationships have been viewed as something to be avoided in order to achieve non-interference [8].

In contrast, we claim that because of the fundamental role of causality in distributed systems, the accurate detection (but not elimination) of causal relationships can be crucial to security in distributed systems. This was first illustrated by Reiter *et al.* [9] by the following example of 'insider trading'. Suppose that a trader issues a request to a trading service to purchase shares of stock, and then as a result of an (indirect or direct) interaction with another trader, the other trader infers that this request has been made. If the latter trader is able to submit a request to the trading service in such a way that the two requests appear to be concurrent, the service could be fooled into processing the latter trader's request first. The result could be that the request of the latter trader could increase the apparent demand for the stock and thus the price offered to the former trader, or enable the latter trader to illegally benefit in a fashion similar to options frontrunning [10]. To prevent these activities, the trading service must recognize that the request of the latter trader is causally after that of the former, and should process that of the former first.

As another example of the importance of causality detection to security, consider a scenario in which a company announces to the trading network that it is merging with another company. Suppose that a trader with inside information of this merger requested to buy large quantities of the company's stock prior to the announcement but, to avoid suspicion, attempted to make it appear that the request was initiated causally after the announcement. If the trading service accepts that the request was initiated causally after the announcement, then the insider trading may go undetected.

More generally, because of the fundamental importance of causality to so many distributed algorithms, the conversion of these algorithms for use in a hostile environment necessarily relies upon the accurate detection of causal relationships despite malicious behaviour. For instance, consider a service that allocates a distributed resource to processes in an order consistent with the causal relationships among their requests [4]. If such a service is to be fair in a hostile setting, it must be able to detect causal relationships accurately, despite attempts of dishonest processes to wrongfully make their requests appear causally prior to other requests.

The above examples show that the type of causality detection required can differ from one circumstance to the next. As shown in the first trading example, it may be required that if a causal relationship exists, then it is detected. On the other hand, in the second example, security relies on an inverse requirement, namely that if a causal relationship is detected, then it actually exists. Thus, depending on the circumstances, it may be important that a principal not be able to deny existing causal relationships or to claim non-existent ones without being detected.

In this paper we formalize possible security goals with respect to causality and present simple algorithms to attain these goals in some situations. This work generalizes and improves upon the treatment of causality by Reiter *et al.* [9] in two ways. First, this work presents a general framework in which attacks on causality can be examined; in this framework, we were able to identify attacks that are not considered by [9]. Second, we present new algorithms to counter these attacks.

The rest of this paper is structured as follows. In Section 2, we describe the assumptions that we make about the system. In Section 3, we formally define the notion of causality. In Section 4 we formalize our security goals with respect to causality. In Sections 5 and 6 we describe several algorithms for reaching these goals. A relationship between causality and the notion of freshness is discussed in Section 7. We summarize and describe related and future work in Section 8.

2. THE SYSTEM MODEL

We assume a system consisting of a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of processes that are spatially separated and that communicate exclusively via a completely connected, point-to-point network.[1] We often denote processes with the letters P, Q, R and S when subscripts are unnecessary. Processes that behave according to their design specification are said to be *correct*. Processes may be *corrupted* and thereafter may exhibit arbitrarily malicious behaviour, limited only by the assumptions stated below.

The execution of each process is modelled as a sequence of indivisible events. There are two types of events that can be executed by processes: sending a message m to a process, denoted by *send*(m), and receiving a message m from a process, denoted by *receive*(m). (Internal computations are not explicitly modelled.) Messages are identified by their send events and not their content, e.g. messages with the same contents sent in different events are different messages for our purposes.

We assume that each process receives only messages that are sent to it (or by it; see below). In particular, communication channels between correct processes are authenticated and protect the integrity and the secrecy of communication, so that corrupt processes cannot tamper with or receive this communication. In addition, all communication between corrupt processes is modelled with explicit sends and receives, regardless of its actual form (e.g. signals via a covert channel). We also assume that channels between correct processes provide FIFO delivery using, for instance, a standard sequence number mechanism [11].

Many algorithms for detecting causality in benign environments utilize assumptions of synchronized clocks or bounded communication delays (e.g. [4]). However, we do not assume that processes maintain synchronized clocks, or that message transmission times between processes or execution speeds of processes are bounded. That is, the system is totally asynchronous.

Finally, to simplify the following discussion, it is convenient to stipulate that at each process, the event *send*(m) is immediately followed by *receive*(m), with no other events occurring between these two. So a message is received by its sender and (possibly) by its intended destination.

3. DEFINITION OF CAUSALITY

We use the notion of causality formulated by Lamport [1]. As described in Section 1, one event is causally before another if it could have affected that other event. More precisely, suppose we define the "one-step" causality relation $\rightsquigarrow$ as the smallest relation satisfying the following conditions:

1. If events e_1 and e_2 are executed consecutively at the same process, then $e_1 \rightsquigarrow e_2$.
2. For any m, $send(m) \rightsquigarrow receive(m)$.

Then, the causality relation $\rightarrow$ is simply the transitive closure of $\rightsquigarrow$.

In this paper, we will be concerned with causal relationships among messages, where two messages are causally related precisely as the events in which they were sent. So, if $send(m_1) \rightarrow send(m_2)$, then we say that m_1 is causally before m_2 and m_2 is causally after m_1. We will often use '$m_1 \rightarrow m_2$' as an abbreviation for '$send(m_1) \rightarrow send(m_2)$'.

Finally, we define a causal chain to be a sequence of events $e_1, e_2, \ldots, e_l$ such that $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_l$. Note that $e_1 \rightarrow e_2$ if and only if there exists a causal chain beginning with e_1 and ending with e_2.

4. CAUSAL SECURITY GOALS

In Section 1 we discussed several examples in which the detection of causal relationships was important for security. In this section we attempt to formulate security goals with respect to causality more carefully. We introduce two notions, denial and forgery, that capture the ways in which efforts to detect causal relationships between messages can fail due to malicious or accidental behavior, and discuss how these notions relate to the examples of Section 1. Sections 5 and 6 are devoted to preventing denial and forgery respectively.

Since there is a version of denial and forgery for each

[1] The results of this paper can be extended for multi-cast communication, although multicast complicates the algorithms and discussion with little benefit. Thus, for simplicity we treat only point-to-point communication here.

causality detection algorithm, when defining these notions it is convenient to abstract all such algorithms as a predicate $\mathcal{C}$ on pairs of messages. That is, we assume that a process determines if message m_1 is causally before message m_2 by evaluating $\mathcal{C}(m_1, m_2)$. If $\mathcal{C}(m_1, m_2)$ evaluates to true, then the process 'believes' that $m_1 \rightarrow m_2$; otherwise, it 'believes' that $m_1 \not\rightarrow m_2$, where $\not\rightarrow$ is the complement of $\rightarrow$. Thus, $\mathcal{C}$ has the following *desired* behaviour:

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } m_1 \rightarrow m_2 \\ \text{false} & \text{otherwise} \end{cases}$$

A correct process P generally need not be able to evaluate $\mathcal{C}$ on all pairs of messages, but should be able to compute $\mathcal{C}(m_1, m_2)$ if both *receive*(m_1) and *receive*(m_2) are executed at P. (Recall that if a process executes *send*(m), then it also executes *receive*(m).) In the remainder of this paper, we concern ourselves only with predicates that a process evaluates on messages it receives.

Given $\mathcal{C}$, we can now define the notions of denial and forgery, which can occur due to malicious or accidental behavior, if $\mathcal{C}$ is not robust to such behaviour.

Denial: A causal relationship is *denied* (with respect to $\mathcal{C}$) if there exist messages m_1 and m_2 such that $m_1 \rightarrow m_2$, but at some correct process $\mathcal{C}(m_1, m_2)$ is false.

Forgery: A causal relationship is *forged* (with respect to $\mathcal{C}$) if there exist messages m_1 and m_2 such that $m_1 \not\rightarrow m_2$, but at some correct process $\mathcal{C}(m_1, m_2)$ is true.

We have already seen examples of how denial and forgery can result in security problems. For instance, reconsider the trading examples in Section 1, which are represented pictorially in Figure 1. In the first example, the second trader Q attempts to deny that its request m_2 is causally after P's request m_1 as a result of its interacting with P (possibly through other processes S). If the attempt is successful, the trading service R may fail to recognize that m_1 should be serviced before m_2. The second example illustrates the dangers of forgery: the trading service R should not interpret the request m_2 from the trader Q to be causally after the announcement m_1 from the company P when in reality it is not.

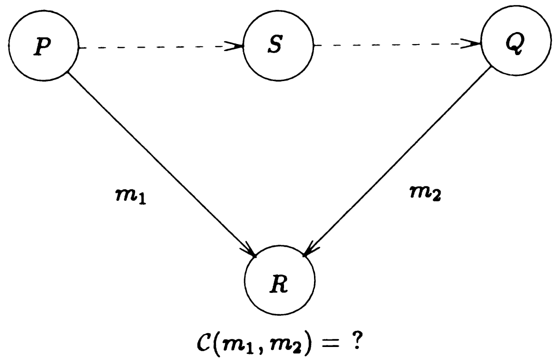
The next two sections of this paper are devoted to finding algorithms to prevent denial or forgery in various situations. In general, to prevent denial it must be the case that

D: If $m_1 \rightarrow m_2$, then $\mathcal{C}(m_1, m_2)$ is true at any correct recipient of m_1 and m_2.

On the other hand, the prevention of forgery requires that precisely the converse hold:

F: If $\mathcal{C}(m_1, m_2)$ is true at any correct recipient of m_1 and m_2, then $m_1 \rightarrow m_2$.

In order to rule out trivial solutions that provide no causal information, we also require that our algorithms



Process R detects causal relationships between messages with the predicate $\mathcal{C}$.

FIGURE 1. Causality detection.

satisfy the following property in addition to preventing denial and/or forgery:

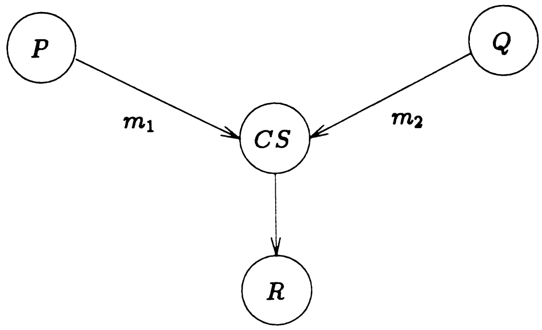
E: If there exists a causal chain $e_1, \ldots, e_l$ such that $e_1 = send(m_1)$, $e_l = send(m_2)$, and for each $j \in \{1, \ldots, l\}$, e_j was executed at a correct process, then at any correct recipient of m_1 and m_2, $\mathcal{C}(m_1, m_2)$ is true and $\mathcal{C}(m_2, m_1)$ is false.

Property E requires that a causal chain traversing only correct processes be recognized, and thus intuitively represents the 'minimum' required of a causality detection algorithm. For our purposes, E serves to rule out some trivial algorithms that provide no causal information, such as '$\mathcal{C}(m_1, m_2)$ = false for all m_1 and m_2' (which satisfies F) and '$\mathcal{C}(m_1, m_2)$ = true for all m_1 and m_2' (which satisfies D).

In Sections 5 and 6 we concentrate on finding algorithms to satisfy E always, and D or F if the sender of m_1 (in the statement of D and F) is correct. In Section 5 we present two algorithms that satisfy E and that satisfy D if the sender of m_1 is correct. Then, in Section 6, we present two algorithms that satisfy E and that satisfy F if the sender of m_1 is correct. What can be done to satisfy D and/or F when the sender of m_1 is corrupt is an open problem. However, the algorithm in Section 6.2 also satisfies a property with only a slightly weaker consequent than F, even if both the senders of m_1 and m_2 are corrupt. We suspect that this property may suffice in some situations.

5. PREVENTING DENIAL

In this section we discuss two methods for preventing denial attacks. More precisely, the algorithms discussed in this section ensure that if a correct process R receives messages m_1 and m_2, where the sender of m_1 is correct and $m_1 \rightarrow m_2$, then $\mathcal{C}(m_1, m_2)$ is true when evaluated at R. So, in the example of Figure 1, these protocols ensure that if m_1 is causally before m_2, then Q cannot 'backdate' m_2 to appear causally before or concurrent with m_1.



The causality server CS forwards messages destined for a process to that process in the order in which it receives those messages.

FIGURE 2. The causality server CS.

5.1. The causality server

Our first solution employs a trusted causality server. Intuitively, the causality server acts as an intermediary between all pairs of processes in the system. Each correct process directly communicates with (i.e. sends messages to or receives messages from) only the causality server, via an authenticated, FIFO channel that protects the integrity and secrecy of communication. For one process to send a message to another process, the former sends it to the causality server. For each process R, the causality server forwards messages destined for R to R, in the order in which the server receives those messages (see Figure 2).

This simple causality server ensures that if processes detect causal relationships with

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } m_1 \text{is received before } m_2 \\ \text{false} & \text{otherwise} \end{cases}$$

then it is not possible for a corrupt process to deny the causal relationships that its messages have with causally prior messages from correct processes.

THEOREM 5.1. *This algorithm satisfies E and satisfies D if the sender of m_1 is correct.*

Proof. D: Suppose there are messages m_1 and m_2 such that $m_1 \rightarrow m_2$ and the sender of m_1 is correct. Also suppose that R is a (correct) recipient of m_1 and m_2. If R is the sender of m_1 (i.e. R sent m_1 to another process), then because m_1 is received at R immediately after it is sent, R receives m_1 before m_2. Now suppose some other process sends m_1 to R. Because the channel from the sender of m_1 to the causality server is FIFO, m_1 must arrive at the causality server before any message m such that $m_1 \rightarrow m$. So, m_1 is forwarded to (and thus is received by) R before any such message destined for R, and in particular, before m_2.

E: Suppose there exists a causal chain $e_1, \ldots, e_l$ such that $e_1 = send(m_1)$, $e_l = send(m_2)$, and for each $j \in \{1, \ldots, l\}$, e_j is executed at a correct process. By the argument for D, $\mathcal{C}(m_1, m_2)$ is true at any correct recipient of m_1 and m_2. Then, because if m_1 is received before m_2 then m_2 is received after m_1, $\mathcal{C}(m_2, m_1)$ is false. □

A warranted concern with the use of a causality server is performance: this scheme results in twice as many messages being transmitted over the network than without the causality server, and the server may become a traffic bottleneck in the system. However, the degree to which a causality server would become a bottleneck might be less than at first expected, because the causality server has very little processing to do on each message it receives and forwards. In fact, in a likely implementation it would simply need to decrypt the message, appropriately check and attach channel sequence numbers (to implement FIFO order), re-encrypt the message, and forward it. Supposing that encryption and decryption can be done in hardware, the performance impact seen by processes could be tolerable.

A second concern with this scheme is that it introduces a single point of failure, namely the causality server, into the system, because all communication would cease if the causality server failed. This problem can be addressed using known replication techniques (e.g. [4]), albeit at an additional cost to performance.

5.2. The conservative approach

An alternative approach to the use of a causality server is for each process P to delay sending a message to its destination until all messages that P previously sent to other destinations have been received at those destinations.[2] In general, a sender can be informed of the receipt of its messages by acknowledgements. These acknowledgements would occur as part of a lower layer protocol, and would not result in additional process events or be delayed like messages.[3] Processes again detect causal relationships with the predicate

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } m_1 \text{is received before } m_2, \\ \text{false} & \text{otherwise.} \end{cases}$$

THEOREM 5.2. *This algorithm satisfies E and satisfies D if the sender of m_1 is correct.*

Proof. D: Suppose there are messages m_1 and m_2 such that $m_1 \rightarrow m_2$ and the sender of m_1 is correct. Also suppose that R is a (correct) recipient of m_1 and m_2. If R is the sender of m_1 (i.e. R sent m_1 to another process), then because m_1 is received at R immediately after it is sent, R receives m_1 before m_2. Now suppose some other process sends m_1 to R. If the same process also sends m_2, then R receives m_1 first because channels between correct

[2] A further condition is required if multicast communication is used (see [4]). However, as stated in Section 2, we restrict ourselves in this paper to point-to-point communication.

[3] These acknowledgements could be viewed as introducing additional causal relationships. However, since acknowledgements carry no application-specific information, these relationships are unlikely to be of interest in most settings and thus are omitted from the present discussion.

processes are FIFO. Otherwise, m_2 can be sent only after m_1 is received at R, because the sender of m_1 does not communicate to destinations other than R until R has received m_1.

E: Suppose there exists a causal chain $e_1, \ldots, e_l$ such that $e_1 = send(m_1)$, $e_l = send(m_2)$, and for each $j \in \{1, \ldots, l\}$, e_j is executed at a correct process. By the argument for D, $\mathcal{C}(m_1, m_2)$ is true at any correct recipient of m_1 and m_2. Then, because if m_1 is received before m_2 then m_2 is received after m_1, $\mathcal{C}(m_2, m_1)$ is false. □

This approach, sometimes called the conservative approach, has been used by several systems to detect causal relationships in benign environments (e.g. [4,5]). It is especially attractive in our setting because a correct process can singlehandedly prevent corrupt processes from 'backdating' their messages to wrongly appear causally prior to or concurrent with its own. That is, it need not rely on a third party for this guarantee. Moreover, this solution introduces no bottleneck or single point of failure into the system.

Communication performance achieved with the conservative approach can vary widely, depending on the particular communication patterns exhibited by processes. Because a process delays sending a message to a destination only when it does not know of the receipt of a message it previously sent to a *different* destination, processes can achieve the full performance benefits of asynchronous communication when streaming messages to a single destination. However, when processes send to many different destinations in quick succession, the communications are essentially reduced to synchronous remote procedure calls.

From a security point of view, the most significant disadvantage of the conservative protocol is the potential for denial-of-service attacks. A corrupt process can prevent a sender of a message from being able to send to any other destinations by refusing to acknowledge any messages sent to it. (This form of 'attack' can occur even in benign environments if a process simply crashes.) Different policies can be implemented to deal with this problem, and which is best depends on the particular system and application. One approach is implemented in the Isis system, which uses a version of the conservative protocol adapted for multicast communication [5, 9]. In Isis, a trusted, fault-tolerant service called the failure detector declares processes faulty when they appear so and excludes them from the system [12]. Using this approach with minor modifications to address malicious process behaviour, it is possible to ensure that a process that attempts denial-of-service attacks by refusing to acknowledge messages will eventually be considered faulty and excluded from the system. Any process waiting for acknowledgements from the excluded process would be allowed to proceed with sending to other processes without jeopardizing causality detection, even if the excluded process was correct but deemed faulty due to network delays.

6. PREVENTING FORGERY

In this section we present two algorithms that satisfy F if the sender of m_1 is correct. That is, they ensure that if a correct process R receives m_1 and m_2, the sender of m_1 is correct, and $m_1 \not\to m_2$, then $\mathcal{C}(m_1, m_2)$ is false when evaluated at R. As discussed in Section 4, satisfying F under only the assumption that the sender of m_2 is correct is an open problem. However, the second algorithm presented here does satisfy a property with only a slightly weaker consequent than F, even if both the senders of m_1 and m_2 are corrupt. We believe that especially in the case in which the sender of m_2 is correct, this property may suffice for some applications.

These algorithms use a digital signature scheme. We assume that each process P_i holds a private key K_i with which it can sign information so that any other process can verify the information's origin and authenticity. Information m so signed is denoted $\{m\}_{K_i}$.

6.1. Signed vector timestamps

Our first algorithm originates from a technique introduced by Lamport [1], where he described an algorithm using logical clocks to detect causal relationships among messages (in benign environments). In his technique, each process P_i maintains a logical clock t_i that assigns a value $t_i[e]$ to each event e executed at P_i, according to the following constraint known as the clock condition:

T1: For any events e_1 and e_2, if $e_1 \to e_2$, then $t_i[e_1] \prec t_j[e_2]$.

(The notation '$t_i[e]$' implies that P_i executed e.)

In Lamport's algorithm, each logical clock t_i was implemented by an integer counter and '$\prec$' was normal integer less than ($<$); thus, it was not possible to attain the converse of the clock condition, as well. Later, however, several researchers (e.g. [13]) extended the notion of logical clocks to that of vector clocks and defined a new relation '$\prec$' on them so that the converse condition could also be satisfied:

T2: For any events e_1 and e_2, if $t_i[e_1] \prec t_j[e_2]$, then $e_1 \to e_2$.

In the algorithm in [13], each process P_i maintains a vector clock $t_i = \langle t_i^1, t_i^2, \ldots, t_i^n \rangle$, where n is the total number of processes in the system and for each $k \in \{1, \ldots, n\}$, t_i^k is a nonnegative integer. Vector clock values $t = \langle t^1, \ldots, t^n \rangle$ and $\hat{t} = \langle \hat{t}^1, \ldots, \hat{t}^n \rangle$ are ordered according to the following relation: $t \prec \hat{t}$ iff for all $k \in \{1, \ldots, n\}$, $t^k \leqslant \hat{t}^k$, and there exists a $k \in \{1, \ldots, n\}$ such that $t^k < \hat{t}^k$. The algorithm to satisfy T1 and T2 is as follows:

1. When process P_i begins execution, t_i is initialized to all zeroes.
2. Process P_i increments t_i^i before executing each event.
3. If $send(m)$ is executed by process P_i, then the timestamp $T_m = t_i$ is sent with m. $t_i[send(m)]$ is defined to be t_i.

4. If *receive*(m) is executed by process P_j, then for all $k \in \{1, \ldots, n\}$, P_j sets t_j^k to $\max\{t_j^k, T_m^k\}$, where T_m^k is the kth component of T_m. $t_j[receive(m)]$ is then defined to be t_j.

Because the timestamp on a message m sent by P_i is $T_m = t_i[send(m)]$ (by step 3), this algorithm can be seen as using the following predicate to determine causal relationships:

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } T_{m_1} \prec T_{m_2} \\ \text{false} & \text{otherwise} \end{cases}$$

In our system model, this algorithm does not suffice to prevent processes from forging causal relationships, because a corrupt process can easily manipulate components of vector timestamps. For instance, in Figure 1, Q could easily fabricate a timestamp T_{m_2} to make m_2 wrongly appear causally after m_1.

We thus propose a technique to prevent this. In our approach, processes maintain vector clocks as before. However, each process P_i digitally signs the ith component of each timestamp it includes with a message, and this signed value is then propagated by other processes in the ith components of the timestamps they include with their messages. So, when a process P_i executes *send*(m), it includes with m a vector timestamp of the form

$$T_m = \langle \{t_i^1\}_{K_1}, \{t_i^2\}_{K_2}, \ldots, \{t_i^n\}_{K_n} \rangle$$

where for each $k \neq i$, $\{t_i^k\}_{K_k}$ was received by P_i in a previous receive event. The requirement that each (non-zero) component of a vector timestamp be signed by the corresponding process prevents corrupt processes from inflating components of correct processes.

More precisely, the algorithm executes as follows:

1. When process P_i begins execution, t_i is initialized to all zeroes.
2. Process P_i increments t_i^i before executing each event.
3. If *send*(m) is executed by process P_i, then the timestamp $T_m = \langle T_m^1, \ldots, T_m^n \rangle$ is sent with m, where for each $k \in \{1, \ldots, n\}$,

$$T_m^k = \begin{cases} 0 & \text{if } t_i^k = 0, \\ \{t_i^k\}_{K_k} & \text{otherwise.} \end{cases}$$

4. If *receive*(m) is executed by process P_j, and for all $k \in \{1, \ldots, n\}$, T_m^k is properly signed by P_k or is zero, then for all $k \in \{1, \ldots, n\}$, P_j sets t_j^k to $\max\{t_j^k, \overline{T_m^k}\}$, where $\overline{T_m^k} = 0$ if $T_m^k = 0$, and $T_m^k = \{\overline{T_m^k}\}_{K_k}$ otherwise. Then, for each $k \in \{1, \ldots, n\}$ such that $t_j^k > 0$, P_j saves $\{t_j^k\}_{K_k}$, which it either received as T_m^k or already had prior to this event. If some non-zero T_m^k is not properly signed by P_k, then because communication channels between correct processes protect the integrity of communication, this message must be from a corrupt process and is therefore ignored.

Note that each T_m^k can always be computed by a correct process P_i in step 3 of this algorithm, because if $k \neq i$ and $t_i^k \neq 0$, then by step 4, $T_m^k = \{t_i^k\}_{K_k}$ was received and saved by P_i in a previous receive event. Processes detect causal relationships between messages with the same predicate as before, adjusted for the signatures:

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } \overline{T_{m_1}} \prec \overline{T_{m_2}} \text{ and } \forall k \in \{1, \ldots, n\}, \text{ each of } T_{m_1}^k \text{ and } T_{m_2}^k \text{is signed by } P_k \text{ or is } 0 \\ \text{false} & \text{otherwise} \end{cases}$$

where $\overline{T_m} = \langle \overline{T_m^1}, \ldots, \overline{T_m^n} \rangle$.

THEOREM 6.1. *This algorithm satisfies E and satisfies F if the sender of m_1 is correct.*

Proof. E: Suppose there is a causal chain $e_1, \ldots, e_l$ such that $e_1 = send(m_1)$, $e_l = send(m_2)$, and for each $j \in \{1, \ldots, l\}$, e_j is executed at a correct process. By construction, each component of T_{m_1} and T_{m_2} is properly signed or zero, and $\forall k \in \{1, \ldots, n\}$, $\overline{T_{m_1}^k} \leqslant \overline{T_{m_2}^k}$ by step 4. Moreover, if the sender of m_2 is P_i, then $\overline{T_{m_1}^i} < \overline{T_{m_2}^i}$ by step 2. So, by the definition of "$\prec$" for vector timestamps, $\overline{T_{m_1}} \prec \overline{T_{m_2}}$ and $\overline{T_{m_2}} \not\prec \overline{T_{m_1}}$.

F: Suppose that a correct process R receives m_1 and m_2, where the sender P_i of m_1 is correct, and that $\mathcal{C}(m_1, m_2)$ is true at R. Then, $\overline{T_{m_1}^i} \leqslant \overline{T_{m_2}^i}$. Moreover, by step 2 of the algorithm, $\overline{T_{m_1}^i} > 0$, and so $T_{m_2}^i$ must be signed by P_i. If P_i sent m_2, then $m_1 \rightarrow m_2$ because $\overline{T_{m_1}^i} \leqslant \overline{T_{m_2}^i}$ implies that P_i sent m_2 after m_1. If another process sent m_2, then there must be a causal chain by which $T_{m_2}^i$ traveled from P_i to the sender of m_2. Because P_i released $T_{m_2}^i$ only with m_1 or a causally subsequent message, it follows that $m_1 \rightarrow m_2$. □

In this algorithm, if P_i is correct, then corrupt processes cannot inflate timestamps' ith components above their proper values, because the signatures for the inflated values are not predictable before P_i releases them. Thus, this technique is similar to the use of nonce identifiers [14], in that causal relationships are established by the presence of 'new', unpredictable, and verifiable values (i.e. the signed components) in messages. However, our algorithm is more powerful because any process can verify each value, and not just the process that issued it. This technique also has other beneficial features; in particular, it requires no centralized servers, and communication can proceed completely asynchronously.

The primary weakness of this algorithm is its ability to scale. As n becomes large, signed vector timestamps could consume significant network bandwidth. Techniques similar to some of those described in [5] for compressing timestamps in benign systems are appropriate for use in our system model but will not be discussed here. A second threat to scale is that the cost of computing and verifying signatures could be significant if n is large. However, a signature scheme with a fast verification algorithm could lessen this burden, because in this use, signatures will typically be verified more frequently than they are created.

6.2. The piggybacking algorithm

Our second algorithm for satisfying F if the sender of m_1 is correct is based on a piggybacking technique that, to our knowledge, was first used in an early version of the Isis system to detect causal relationships in benign settings [15]. This algorithm is more costly than that in Section 6.1. However, it is interesting because it also satisfies the following property (which is slightly weaker than F), even if both the senders of m_1 and m_2 are corrupt:

F′: If $C(m_1, m_2)$ is true at any correct recipient of m_1 and m_2, then there exists a message m_3 with the same contents as m_1 such that $m_3 \rightarrow m_2$.

Note that this property does not ensure that $m_1 \rightarrow m_2$, but only that some message identical to m_1 causally precedes m_2. While F′ holds with no assumptions on the senders of m_1 and m_2, it is primarily of interest in the case in which the sender of m_2 is correct. In this case, F′ can substantially limit what a corrupt process can choose for the contents of m_1 once m_2 is sent (if $C(m_1, m_2)$ is to be true). Moreover, we will describe additions to our algorithm that place even greater restrictions on the contents of m_1.

Intuitively, the algorithm is very simple. When a process P sends a message m, it 'piggybacks' on (i.e., includes with) m a set H_m of all messages that P received in the past and the messages piggybacked on them. This is illustrated in Figure 3, where P sends m_1 to R and then m to Q, and then Q sends m_2 to R. A process that receives two messages m_1 and m_2 considers m_1 to be causally before m_2 only if (a message with the same contents as) m_1 appears in H_{m_2}

More precisely, the algorithm executes as follows:

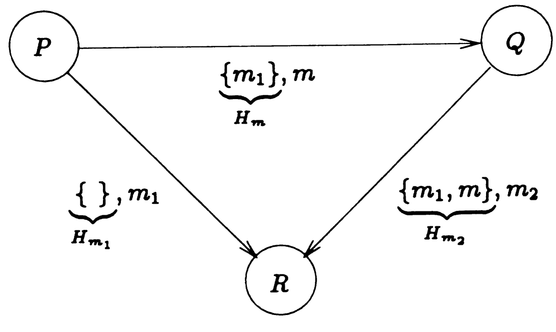
1. Each process P_i maintains a set h_i that is initially empty.
2. If P_i executes $send(m)$, $H_m = h_i$ is sent with m.
3. If P_j executes $receive(m)$, it sets h_j to
$$h_j \cup H_m \cup \{m\}.$$

Processes detect causal relationships as follows:

$$C(m_1, m_2) = \begin{cases} \text{true} & \text{if } m_1 \in H_{m_2}, \\ \text{false} & \text{otherwise.} \end{cases}$$

Here, '$m_1 \in H_{m_2}$' means that a message with contents identical to m_1 appears in H_{m_2}.

While the algorithm already satisfies F′, additional measures must be taken to satisfy E and to satisfy F if the sender of m_1 is correct. To satisfy F under only the assumption that the sender of m_1 is correct, it must not be possible for the sender of m_2 to include (the contents of) m_1 in H_{m_2} unless $m_1 \rightarrow m_2$. That is, m_1 must be unpredictable. In addition, to satisfy E, the contents of messages sent by correct processes must be unique. To



Here P sends m_1 to R and then sends m to Q, piggybacking m_1 on m. Then, Q piggybacks m_1 and m on message m_2 to R.

FIGURE 3. The piggybacking algorithm.

see why, suppose there exist messages m_1 and m_2 such that m_1 causally precedes m_2 by means of a causal chain traversing correct processes only. If the sender of m_2 had previously sent a message whose contents were identical to those of m_2, then this message could appear in H_{m_1}, causing $C(m_2, m_1)$ to be true at a correct recipient of m_1 and m_2.

One way to make correct processes' messages unique and unpredictable is for the kth message m from P_i to P_j to be constructed in the form '$\{i, j, k : data\}_{K_i}$' where data denotes the data to be sent in the message (not including H_m). Specifying i, j and k in the message makes the message contents unique, and including the signature makes the message contents unpredictable. Then, we can prove

THEOREM 6.2. *This algorithm satisfies* E *and* F′, *and satisfies* F *if the sender of* m_1 *is correct.*

Proof. E: Suppose there exists a causal chain $e_1, \ldots, e_l$ such that $e_1 = send(m_1)$, $e_l = send(m_2)$, and for each $j \in \{1, \ldots, l\}$, e_j is executed at a correct process. By construction, $m_1 \in H_{m_2}$; so, $C(m_1, m_2)$ is true. In addition, since the signature in m_2 first appeared when m_2 was sent and because $m_2 \not\rightarrow m_1$, m_2 could not be in H_{m_1}.

F′: Suppose that a correct process R receives m_1 and m_2, and that $C(m_1, m_2)$ is true at R. Then, $m_1 \in H_{m_2}$. Consider any causal chain $e_1, \ldots, e_l$ of maximum length such that $e_1 = send(m)$ for some m, $e_l = receive(m_2)$ at R, and $m_1 \in H_m$. Such a chain exists because, e.g., the chain $send(m_2) \rightsquigarrow receive(m_2)$ satisfies these requirements. Then, there is some message m' identical to m_1 such that $receive(m')$ was executed at the sender of m before $send(m)$.[4] So, $m' \rightarrow m_2$.

F: Suppose that a correct process R receives m_1 and m_2, the sender P of m_1 is correct, and $C(m_1, m_2)$ is true at

[4] Strictly speaking, the sender of m, if corrupt, could have created m' and included m' in H_m, without receiving m'. For all practical purposes, however, this can be modelled as it sending m' to itself (and thus receiving m') before sending m.

R. Then, $m_1 \in H_{m_2}$. If P sent m_2, then because each message sent by P is unique, $m_1 \in H_{m_2}$ implies that $m_1 \to m_2$. If another process sent m_2, then $m_1 \to m_2$ because the contents of m_1 cannot be predicted by the sender of m_2. □

As mentioned earlier, F′ is of interest primarily when the sender of m_2 is correct (and thus does not cooperate with the sender of m_1 to forge causal relationships). To see why, suppose that a corrupt process P intends to send a message m_1, $m_1 \not\to m_2$, so that $\mathcal{C}(m_1, m_2)$ is true at a correct common destination R. F′ dictates that P choose the contents of m_1 from those messages m_3 such that $m_3 \to m_2$. If m_2 has not yet been sent, P could try to predict its possible choices for m_1 and send these to the sender of m_2. Once m_2 is sent, however, P's choices are limited.

Moreover, by adding some additional checking to our algorithm, we can further narrow the choices available for the contents of m_1. Note that after receiving m_1 (on the channel from P) and m_2, R can detect if

- the sender and receiver listed in m_1 are not P and R, respectively;
- m_1 is the kth message that R received from P but the sequence number listed in m_1 is not k;
- m_1 is not properly signed by P; or
- there are multiple (non-identical) messages in H_{m_2} listing the same sender, receiver, and sequence number as m_1 and bearing P's signature.

Suppose that R defines $\mathcal{C}(m_1, m_2)$ to be false if any of these hold (and thus P is corrupt), even if $m_1 \in H_{m_2}$. Then, once m_2 is sent, P has at most one choice for the contents of each message m_1 it sends on its channel to R that will make $\mathcal{C}(m_1, m_2)$ true at R.

Several improvements to this algorithm can be made in practice. First, instead of piggybacking H_m on each message m, a process need only piggyback those messages in H_m not piggybacked on a prior message to the same destination. If the destination maintains messages piggybacked from each sender, then H_m can be reconstructed when m is received. Second, in some cases it may be appropriate to forgo transmitting a message separately if it will eventually reach its destination piggybacked on another message. However, this delays the former message to be received no earlier than the latter.

A third improvement, which is incompatible with the second, uses message digests to limit the size of piggybacked messages. A message digest algorithm (e.g. [16]) produces a fixed length message digest from an input of arbitrary length, in such a way that it is computationally infeasible to produce any input having a prespecified target message digest, or to produce two inputs having the same message digest. So, for all practical purposes, a message digest uniquely identifies an input. Using a message digest algorithm f, the algorithm can be modified to limit the length of piggybacked messages as follows:

1. Each process P_i starts with h_i initially empty.
2. If P_i executes *send*(m) and m is P_i's kth message to P_j, then $H_m = h_i$ and $D_m = \{i, j, k : f(m)\}_{K_i}$ are sent with m.
3. If P_j executes *receive*(m), it sets h_j to
$$h_j \cup H_m \cup \{D_m\}$$

The predicate to detect causal relationships becomes

$$\mathcal{C}(m_1, m_2) = \begin{cases} \text{true} & \text{if } \{i,j,k : f(m_1)\}_{K_i} \in H_{m_2} \\ & \text{where } m_1 \text{is of the form} \\ & \{i,j,k : data\}_{K_i}, \\ \text{false} & \text{otherwise} \end{cases}$$

The four previously mentioned checks on m_1 and H_{m_2} can also be employed in this new algorithm. Moreover, it is not difficult to verify that while the item D_m created during a *send*(m) must contain a signature, the message m does not. So, this algorithm can be optimized further by not requiring a signature on m and removing the third of the four additional checks enumerated above.

Other possible improvements include garbage collecting messages from the h_i's (at the cost of sacrificing E in some cases), when causal relationships involving those messages are no longer of interest.

7. ON CAUSALITY AND FRESHNESS

Forgery prevention has applications to detecting freshness, a property that has been examined extensively by the security community. A message is fresh in a run of a protocol if its contents have not appeared in another message sent before this run of the protocol began [17, 18]. Freshness is most commonly studied in the context of cryptographic protocols in order to prevent replay attacks, in which an intruder replays messages from previous runs of the protocol in hopes of convincing the participants, e.g. to accept now-obsolete cryptographic keys. A predominant technique for detecting freshness in cryptographic protocols uses challenge-response techniques in conjunction with nonce identifiers [14]: P challenges Q with a new nonce identifier, which Q must include in its response to P. The appearance of the nonce identifier in the response convinces P that Q's response is fresh.

This method for detecting freshness can be generalized using the notion of causality, in the following way. Given a predicate $\mathcal{C}$ that satisfies F, the freshness of a message m_2 can be detected according to the following rule: if m_1 is considered fresh and $\mathcal{C}(m_1, m_2)$ is true, then m_2 should be considered fresh. The use of nonce identifiers as described above is simply an instance of this rule, in which the nonce identifiers are used to implement $\mathcal{C}$. That is, the appearance of the nonce identifier in Q's response to P is the condition on which $\mathcal{C}$ will indicate that Q's message is causally after P's.

This relationship between freshness and causality, which we earlier noted in the preliminary version of this paper [19], was later (independently) presented in a somewhat different framework by Yahalom [20]. In that work, a related but different notion of freshness is studied, in which freshness is tied to the duration of time between two events. That is, a message is considered fresh if its sending preceded its receipt by at most a prespecified number of ticks on the recipient's local clock. In order to detect this condition, Yahalom argues that a means of verifying precedence and succession between events—or in our parlance, detecting causal relationships between events in a way that prevents forgery—is both necessary and sufficient.

8. SUMMARY AND DISCUSSION

In this paper we have attempted to formalize the problems with detecting causality in hostile environments and to provide algorithms to overcome these problems in some situations. In particular, we have introduced two new notions—*denial* and *forgery*—that capture the ways in which causality can be mistakenly not detected or detected. We have presented two algorithms for preventing denial and two algorithms for preventing or limiting forgery in some situations.

Our previous studies of detecting causality in hostile environments include a research effort directed at building secure distributed systems [9]. As part of that effort, a variant of the conservative protocol of Section 5.2 has been implemented. One direction for future work is the implementation of other algorithms so that comparisons between them can be made in real systems. Other directions for future work are discussed in Section 8.2.

8.1. Related work

Smith and Tygar [21] have also examined the detection of causal relationships in hostile environments. In their work, they have similarly identified the detection of causal relationships to be important in hostile environments and have pursued ways of detecting causal relationships despite malicious behavior. In particular, they independently discovered a protocol similar to that of Section 6.1 of this paper [21].

Despite these similarities, however, the works are also substantially different. Smith and Tygar arrived at a different formulation of the problem of detecting causal relationships and, to our knowledge, developed only an approach to preventing (what we call) forgery, namely the technique of signed vector timestamps described in Section 6.1 of this work. On the other hand, while we have studied causality detection only as an essential aspect of preserving integrity in distributed systems, they also raise consideration of the ramifications of causality detection on secrecy requirements. In particular, they introduce notions of forward confinement and backward confinement, which characterize requirements that information about a process not leak to other processes executing events causally after or before its own, respectively. Subsequent work by Smith and Tygar [10] examines the application of trusted hardware coprocessors to these problems.

In other related work, Reiter [22] addresses the issue of preventing causal denial specifically among requests issued to a replicated service, some of whose clients and servers may have been corrupted by an intruder. The approach presented there does not fully prevent denial in the sense of the present paper, but nevertheless may suffice for many applications.

8.2. Future work

There are several directions for future work that we hope to pursue. The most obvious is to find better algorithms to detect causal relationships. In particular, what can be done toward satisfying D or F if the sender of m_1 is corrupt should be examined more closely. Less general algorithms that exploit knowledge of communication patterns are also of interest, especially if applicable to large classes of distributed algorithms.

Another direction for future research is to explore the degree to which patterns of communication must be restricted to prevent denial and forgery in certain situations. It is interesting to note that both of our algorithms for preventing denial synchronize communication, in that they eliminate all executions in which there are messages m_1 and m_2 such that the sender of m_1 is correct, $m_1 \rightarrow m_2$, and yet m_2 is received before m_1 at a correct common destination. On the contrary, neither of our algorithms for preventing forgery restrict patterns of communication at all. We suspect that these are not properties of our algorithms alone, but suggest requirements inherent in the problems.

Finally, another difficult problem is how a process P can determine whether it has received all messages sent to it that are causally prior to a certain received message. Such determinations are necessary if, e.g. P must deliver received messages to an application in an order consistent with the causal relationships among them (e.g. [4, 5]). The algorithms of Section 5 ensure that all causally prior messages have been received if all such messages are sent by correct processes, but this does not necessarily hold if a causally prior message is sent by a corrupt process.

ACKNOWLEDGEMENTS

We are very grateful to Tushar Chandra for suggesting the idea of piggybacking, which led us to the algorithm of Section 6.2. Comments made by Ken Birman, Brad Glade, Andre Schiper, Robbert van Renesse and anonymous referees improved our presentation. A preliminary version of this paper appeared in the *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*. The first author was partially supported under DARPA/ONR grant N00014-92-J-1866, and by grants from GTE, IBM, and Siemens, Inc. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views or decisions of the ONR.

REFERENCES

[1] Lamport, L. (1978) Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7), 558–565.

[2] Peterson, L. L., Buchholz, N. C. and Schlichting, R. D. (1989) Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, **7**(3), 217–246.

[3] Herlihy, M. P. and Wing, J. M. (1990) Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3), 463–492.

[4] Schneider, F. B. (1990) Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, **22**(4), 299–319.

[5] Birman, K. P., Schiper, A. and Stephenson, P. (1991) Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, **9**(3), 272–314.

[6] Ladin, R., Liskov, B., Shrira, L. and Ghemawat, S. (1992) Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, **10**(4), 360–391.

[7] Birman, K. P., Cooper, R. and Gleeson, B. (1991) *Design Alternatives for Process Group Membership and Multicast*. Technical Report 91-1257, Department of Computer Science, Cornell University.

[8] Goguen, J. A. and Meseguer, J. (1982) Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 11–20.

[9] Reiter, M. K., Birman, K. P. and Gong, L. (1992) Integrating security in a group oriented distributed system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, 18–32.

[10] Smith, S. W. and Tygar, J. D. (1994) Security and privacy for partial order time. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems.*

[11] Voydock, V. L. and Kent, S. T. (1983) Security mechanisms in high-level network protocols. *ACM Computing Surveys*, **15**(2), 135–171.

[12] Ricciardi, A. M. and Birman, K. P. (1991) Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 341–351.

[13] Mattern, F. (1989) Virtual time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V. (North-Holland) 215–226.

[14] Needham, R. M. and Schroeder, M. D. (1978) Using encryption for authentication in large networks of computers. *Communications of the ACM*, **21**(12), 993–999.

[15] Birman, K. P. and Joseph, T. A. (1987) Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, **5**(1), 47–76.

[16] Rivest, R. L. (1991) The MD4 message digest algorithm. In Menezes, A. J. and Vanstone, S. A., editors, *Advances in Cryptology—CRYPTO '90 Proceedings* (Lecture Notes in Computer Science 537), Springer-Verlag, 303–311.

[17] Burrows, M., Abadi, M. and Needham, R. (1989) *A Logic of Authentication.* Technical Report 39, Digital Equipment Corporation Systems Research Center.

[18] Abadi, M. and Tuttle, M. R. (1991) A semantics for a logic of authentication. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 201–216.

[19] Reiter, M. K. and Gong, L. (1993) Preventing denial and forgery of causal relationships in distributed systems. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, 30–40.

[20] Yahalom, R. (1993) Optimality of multi-domain protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 38–48.

[21] Smith, S. and Tygar, J. D. (1993) Signed vector timestamps: A secure protocol for partial order time. Technical Report CMU-CS-93-116, School of Computer Science, Carnegie Mellon University.

[22] Reiter, M. K. and Birman, K. P. (1994) How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, **16**(3), 986–1009.