Security and Privacy for Partial Order Time[*]

Sean W. Smith J.D. Tygar
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

July 1994

Abstract

Partial order time expresses issues central to many problems in asynchronous distributed systems, but suffers from inherent security and privacy risks. Secure partial order clocks provide a general method to develop application protocols that transparently protect against these risks. Our previous *Signed Vector Timestamp* [32] protocol provided a partial order time service with some security: no one could forge dependence on an honest process. However, that protocol still permitted some forgery of dependence, permitted all denial of precedence, and leaked private information. This paper uses *secure coprocessors* to improve the vector protocol: our new *Sealed Vector Timestamp* protocol detects both the presence and absence of precedence even in the presence of malicious processes, and protects against some privacy risks as well. Our new protocol solves previously open security problems, and provides a foundation for incorporating security and privacy into distributed application protocols based on partial order time.

Keywords: Distributed systems, privacy, security, secure coprocessor.

1 Introduction

Motivation Partial order time [9, 17, 21, 24, 40] is central to solving application problems in asynchronous distributed systems. Explicitly providing a partial order time service simplifies and clarifies the task of protocol design for applications including *snapshots and global states* [5, 19, 22], *deadlock detection* [16, 20, 34], *immediate ordered service* [15], and *optimistic rollback recovery* [14, 23, 31, 33].

However, while real time can be determined from an independent physical device, partial order time cannot be determined in isolation. Tracking partial order time requires collecting and sharing information. Consequently, partial order time exposes protocols to security risks. Is the information a process receives correct? Can shared information be used for dishonest purposes?

Encapsulating a system's dealings with partial order time into a single time service provides an arena to examine and resolve security and temporal issues for protocol design. (We develop this material in [30].)

This Paper In this paper, we use new developments in inexpensive tamper-proof hardware to build the *Sealed Vector Timestamp* protocol, which provides stronger security and privacy protection than any previous protocol. Sealed Vectors solve previously open problems from [26] by preventing dishonest processes from forging dependence on *any* events, and by preventing dishonest processes from denying dependence (if malicious processes cannot communicate covertly). (Even with covert communication, Sealed Vectors provide some protection against denying dependence.) Sealed Vectors also move beyond previous work by addressing privacy risks, and by providing secure clocks for partial orders where information flow does not imply precedence.

Section 2 reviews partial order time. Section 3 discusses the inherent security and privacy risks. Section 4 surveys the defenses and presents our new protocol. Section 5 discusses our new protocol and considers some directions for future research.

[*]This research was sponsored in part by the Advanced Research Projects Agency (under contract number F19628-93-C-0193), IBM, Motorola, the National Science Foundation (under Presidential Young Investigator Grant CCR-8858087), TRW, and the U. S. Postal Service. The first author also received support from an ONR Graduate Fellowship. As of October 1994, the first author will be affiliated with the Computer Research and Applications Group at the Los Alamos National Laboratory, Los Alamos NM 87545.

2 Clocks for Partial Order Time

Partial order time (POT) is the major alternative time model to global sequential time. Suppose our asynchronous system consists of a collection of n processes (each of which experiences a linear sequence of events), real time clocks do not exist, and the duration between consecutive events at a process is unpredictable. Processes communicate by passing messages that arrive at most once after an unpredictable delay.

We represent the behavior of this system as the *POT* directed graph. For each event, construct a node; for each process, draw edges connecting consecutive events; for each received message, draw an edge from the *send* event to the *receive* event. The transitive closure $\overline{\text{POT}}$ determines a partial order on events. We write $A \longrightarrow B$ to indicate that event A precedes event B in this order; we write $A \Longrightarrow B$ when A precedes B, or A and B are the same event; we write $A \not\longleftrightarrow B$ to indicate that A and B are incomparable under the order. Incomparable events are *concurrent*: neither event could have influenced the other.

We consider clocks for partial order time that allow processes to determine the precedence (or concurrency) of events during a computation. At some event C, process p sends the query $\mathsf{Precedes}(A, B)$ to its clock to learn whether or not event A precedes event B in the partial order. A natural way to build clocks for partial orders where precedence implies information flow is to use timestamps. When an event A occurs, a packet of data is generated comprising the *timestamp* $T(A)$ of event A. The timestamp is passed along with the event name, and carries sufficient information to sort the event relative to other events.

Vector Timestamps are a well-known method for partial order clocks [8, 20, 33]. Each process maintains a local event counter, and timestamps each event A with a vector $\mathbf{V}(A)$. This vector contains one entry for each process. The process q entry of $\mathbf{V}(A)$ is the index of the maximal q event that precedes or equals A. We adopt the convention that a global initial event $\perp$ precedes all other events.

The linear ordering of events at each process suggests a natural ordering on vector timestamps: for vectors V and W, we say that V precedes W (written $V \prec W$) if each entry of V precedes or equals the corresponding entry of W, but $V \neq W$.

It follows directly that vector timestamps function as clocks.

Theorem 1 $\forall A, B \quad \mathbf{V}(A) \prec \mathbf{V}(B) \Longleftrightarrow A \longrightarrow B$

Vector timestamps are easy to implement. Each process p maintains a vector V for its most recent event. When a new event occurs, the process increments the p entry of V. If this new event is a *send*, process p appends the vector V to the message. If this new event is a *receive*, process p reads the timestamp W from the message and replaces V with the entry-wise maximum of V and W.

3 Security and Privacy Risks

Partial order time draws on data distributed throughout the system. Consequently, building partial order clocks requires that processes share private information, and trust the private information shared with them. This opens opportunities for Byzantine (malicious) processes to manipulate the clock protocols, and consequently to manipulate application protocols built on these clock protocols.

Nonsense Attacks Malicious processes can send arbitrary vector entries. Since honest processes will dutifully copy and pass on these values, a single act by a single malicious process can destroy the validity of many vectors throughout the system. (Lamport total order clocks [17] are particularly vulnerable to these attacks.) Simple sanity checks fail to combat this problem. Suppose vector entries are integers. If honest processes refuse to accept vector entries that have increased more than N, a dishonest process can repeatedly increase an entry by $N-1$. The next honest process the victim talks to may then mistakenly identify the honest victim as corrupt.

Malicious Backdating Malicious processes can selectively reduce vector entries, and thus fool honest processes into thinking events happened earlier than they really did. Consider the application of trading commodities options on a public network. Figure 1 shows how *Malicious Backdating* permits the crime of *options frontrunning*, which can occur when brokers may trade both for themselves and for their clients. (One place where options frontrunning occurs is the Chicago commodities exchange.) If a broker happens to buy a small quantity of shares for himself before his client requests a large number of shares, then the broker will make a tidy sum. Consequently, on receiving a client request, a dishonest broker has incentive to issue a request of his own that appears not to have followed the client request. In an electronic exchange using vector clocks, a malicious broker can do this by re-using an old vector on his purchase request.[1]

[1] In the physical Chicago exchange, the only defense the FBI has against options frontrunning is placing undercover agents in the pit to look for unusually lucky brokers.

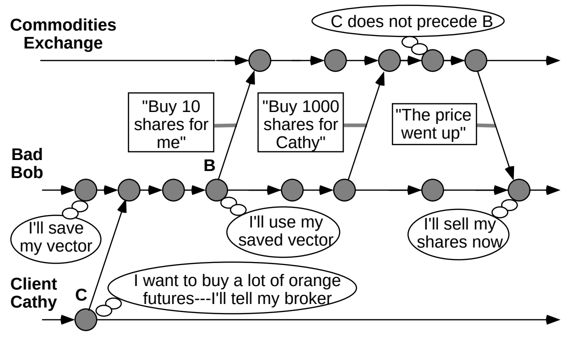

Figure 1: Malicious processes can selectively backdate nodes. Here, *Bob* commits the crime of *options frontrunning* by making his own purchase appear not to follow his client's request.

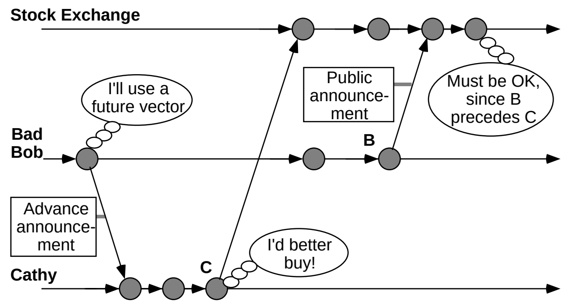

Figure 2: Malicious processes can selectively postdate nodes. Here, *Bob* leaks an advance copy of his public announcement to *Cathy* in such a way that allows her to act on the data first, without appearing to have had a headstart.

Malicious Postdating Malicious processes can selectively inflate vector entries, and thus fool honest processes into thinking events happened later than they really did. Figure 2 shows how such *Malicious Postdating* permits *insider trading*. A malicious process can send a cohort an advance copy of an announcement *along with an advanced vector*. The cohort can act on this data, but use the advanced vector to hide her headstart. (The cohort could even be unwitting; the malicious process might frame her now, in order to spread the blame should the ruse be discovered later.)

Compromised Privacy Malicious processes can correctly perform the vector clock protocol, but use the vector entries to gain illicit knowledge. Figure 3 shows how this technique reveals anonymous whistleblowers. Changes in subsequent timestamp vectors sent from *Alice* to *Bob* show the identities of processes communicating with *Alice*.

4 Defenses

An ideal clock should report "$A \longrightarrow B$" exactly when A precedes B, even if processes perform malicious actions. An ideal clock should also confine private information. We can evaluate clock protocols by this standard: against decreasing amounts of honesty, how well do clocks perform?

Many application protocols use forms of partial order time and vector clocks. A clock protocol meeting this ideal transparently protects higher-level applications against the security and privacy risks of Section 3.

4.1 Previous Work

If all processes are honest, then the process p entries in all vector timestamps originate at process p. The *Signed Vector Timestamp* protocol [26, 32] builds on this observation by requiring each process to digitally sign its entries in outgoing timestamp vectors. This scheme prevents malicious processes from advancing vector entries belonging to honest processes. If an event A occurs at an honest process and our time model expresses all information flow paths, then possession of a signed entry for A is proof of dependence on A. With Signed Vectors, $A \longrightarrow B$ when an honest clock reports "$A \longrightarrow B$" (and A occurs at an honest process). If all processes along a precedence path from A to B are honest, the converse is also true: an honest clock reports "$A \longrightarrow B$" when $A \longrightarrow B$.

However, Signed Vectors may fail if precedence paths go through malicious processes. For example,

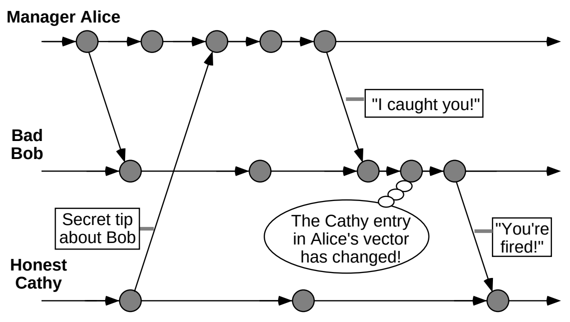

Figure 3: Malicious processes can exploit vector data for illicit purposes. Here, *Bob* uses the timestamp vectors from *Alice* to learn the identity of whistleblower *Cathy*.

a malicious process can use old values in the vector entries for honest processes, as long as the malicious process has retained the matching signatures. Signed Vectors still permit the Malicious Backdating and Malicious Postdating attacks. Signed Vectors do not even attempt to address the Compromised Privacy attack. These problems migrate to higher-level applications. Inability to detect non-precedence reliably can result in inefficiency (in *optimistic rollback recovery*, processes may mistakenly believe they depend on failed states) or complete incorrectness (in *global state* protocols, processes may make incorrect decisions regarding "concurrent" events).

The security of the Signed Vector protocol depends on the fact that precedence paths and information flow paths coincide. If precedence and information flow do not coincide, then Signed Vectors do not provide secure clocks. For example, consider the partial order describing the virtual computation arising after rollback with modified replay.

Three additional protocols exist for the special case of a process sorting the *send* events of two messages it has received [26]. The *Piggybacking* protocol generalizes the vector timestamp protocol by timestamping each event E with a signed record of all messages whose *send* events precede E. Piggybacking (like Signed Vectors) ensures that if a clock reports "$A \longrightarrow B$" and A occurs at an honest process, then $A \longrightarrow B$; Piggybacking further limits the possible actions of a dishonest A process conspiring to make a clock falsely report "$A \longrightarrow B$." However, the Piggybacking protocol (also like Signed Vectors) cannot reliably detect precedence paths touching malicious processes, and does not address the issue of privacy. The other two protocols from [26] alter the order in which messages are received. These protocols address the problem of detecting the partial order by changing the partial order; further, they do not accurately report non-precedence. The *Conservative* protocol requires that before sending a new message, a process wait for acknowledgements of any previous messages it sent. The *Causality Server* protocol assumes secure FIFO channels, and relies on a trusted central intermediary to impose a total order on all message traffic.

4.2 The Sealed Vector Timestamp Protocol

The *Sealed Vector Timestamp* protocol has security properties that solve previously open problems:

- Our protocol accurately reports "$A \longrightarrow B$" or "$A \not\longrightarrow B$," in the presence of arbitrary malicious processes (including the A process).
- Our protocol does not leak private information.

The Sealed Vector Timestamp protocol satisfies the ideal (assuming no covert channels), and protects privacy of vector entries as well. Further, this protocol extends to time models where information flow does not imply precedence. Figure 4 compares our new protocol to previous work.

4.2.1 Overview

Our new protocol rests on the the technology of *secure coprocessors* [35, 41]: inexpensive physically secure devices with a CPU, ROM, and non-volatile RAM. A host processor interacts with its secure coprocessor through formal I/O channels. Any other method of determining the internal state of the coprocessor—including physically penetrating the hardware—results in the erasing of RAM and CPU registers. Secure coprocessors are being deployed rapidly; commercial secure coprocessor products are available from IBM (μABYSS [37], Citadel [39]), and have been announced by other vendors including National Semiconductor [36], Semaphore, Telequip, and Wave Systems. Various protection technologies exist. IBM wraps circuit boards in nichrome wire and then seals them with an epoxy mixture chemically stronger than the wire. A detection circuit monitors the resistance of this wire wrapping; penetration attempts will disrupt the wire wrapping and alter the resistance (e.g., by shorting the wire or by cutting it).

Secure coprocessors only possess *limited* amounts of power. We cannot secure an entire workstation—even if we could, we could not secure the user. Bootstrap-

who's honest?	"$A \to B$" $\Rightarrow$ $A \to B$	"$A \to B$" $\Leftarrow$ $A \to B$	"$A \to B$" $\Leftrightarrow$ $A \to B$	privacy?
path from A to B	Signed, PB, Sealed	Signed, PB, Sealed	Signed, PB, Sealed	Sealed
only A	Signed, PB, Sealed	Sealed	Sealed	Sealed
no one (but you)	Sealed	Sealed	Sealed	Sealed

Figure 4: This table compares how, against decreasing amounts of honesty, partial order clock protocols meet the clock ideal: reporting "$A \longrightarrow B$" $\iff A \longrightarrow B$ while protecting the privacy of vector entries. *Signed* denotes the Signed Vector Timestamp protocol; *Sealed* denotes the Sealed Vector Timestamp protocol; *PB* denotes the Piggybacking protocol.

ping from this small amount of physical security into full protocol security raises subtle issues. For example, malicious processes might attempt to bypass coprocessors, or to attack communication lines. (Recent work [35, 41] shows how to protect against these attacks.)

In the Sealed Vector Timestamp protocol, each process runs on a host processor with a secure coprocessor. The secure coprocessor creates timestamp vectors and *seals* them so that processes cannot read them. Although processes can store and exchange timestamps, they need to query a secure coprocessor in order to compare them.

The security of Sealed Vectors follows from a number of properties. First, no party (except a secure coprocessor) can obtain information about the contents of any vector entry from a sealed timestamp, even if the party knows the other entries. Second, all processes must route incoming and outgoing messages through secure coprocessors. Third, a secure coprocessor must be able to verify that a timestamp was properly sealed by another secure coprocessor. Finally, given a sealed timestamp and an event, a secure coprocessor must be able to verify that they match.

Cryptographic Tools We use *digital signatures* and *bit-secure public key cryptography* [7, 27]. A digital signature is a function S from a value space to a signature space such that:

- Given a value v and a signature s, any party can determine whether s is a valid signature of v: whether $S(v) = s$.
- However, it is intractable for any party (except the privileged signing party) to take a set of value-signature pairs and produce a pair not in this set.

Public key cryptography consists of a function E (from the plaintext space to the cipherspace) and a function D (from the cipherspace to the plaintext space) such that:

- For any plaintext value v, any party can calculate $E(v)$.
- For any plaintext value v, $D(E(v)) = v$.
- It is intractable for any party (except for the privileged decrypting party) to take a set of plaintext-ciphertext pairs and produce a pair not in this set.

Standard public key cryptography requires only that inverting E is difficult (without the privilege of knowing D). Bit-secure public key cryptography requires an additional level of security. Roughly speaking, from a given ciphertext, a malicious process should gain no information about the plaintext that it did not know *a priori*. (See [10] for formal definitions.) Some popular cryptosystems (like [25] and [27]) are known to leak number-theoretic properties of the plaintexts and thus fail to meet this condition [2, 18]. For the Sealed Vector protocol to attain its full security potential, it should be implemented using strong cryptosystems such as [4] or [11].

4.2.2 Operation

We use cryptography and signatures both on messages ($E_{\text{msg}}, D_{\text{msg}}$ and S_{msg}) and on timestamps ($E_{\text{tst}}, D_{\text{tst}}$ and S_{tst}).[2] Each process p has a name, which we denote as p. Each process p runs on a host processor with a secure coprocessor, which we denote as p_{SC}. Each secure coprocessor knows that name of its process.

Let $\mathcal{P}$ be the set of process names, $\mathcal{E}$ be the set of event names, $\mathcal{V}$ be the set of possible timestamp vectors, and $\mathcal{M}$ be the set of possible message texts. Let $\mathcal{G}_{\text{msg}}$ and $\mathcal{G}_{\text{tst}}$ be the signatures spaces for messages and timestamps, respectively; let $\mathcal{C}_{\text{msg}}$ and $\mathcal{C}_{\text{tst}}$ be the cipherspaces for messages and timestamps. Our signature and encryption functions act as follows:

$$\begin{aligned} S_{\text{tst}} &: \mathcal{E} \times \mathcal{V} \mapsto \mathcal{G}_{\text{tst}} \\ E_{\text{tst}} &: \mathcal{E} \times \mathcal{V} \times \mathcal{G}_{\text{tst}} \mapsto \mathcal{C}_{\text{tst}} \\ S_{\text{msg}} &: \mathcal{P} \times \mathcal{P} \times \mathcal{M} \times \mathcal{C}_{\text{tst}} \mapsto \mathcal{G}_{\text{msg}} \\ E_{\text{msg}} &: \mathcal{P} \times \mathcal{P} \times \mathcal{M} \times \mathcal{C}_{\text{tst}} \times \mathcal{G}_{\text{msg}} \mapsto \mathcal{C}_{\text{msg}} \end{aligned}$$

The functions E_{msg} and E_{tst} are public. Each secure coprocessor p_{SC} has the ability to calculate D_{msg}, D_{tst}, S_{msg}, and S_{tst}; the coprocessor p_{SC} also maintains the current process p timestamp vector, which we denote as V_p.

Obtaining Timestamps Suppose process p wants to obtain a timestamp for its current event A. Process p submits the request to p_{SC}, which obtains $\mathbf{V}(A)$ by incrementing the p entry of V_p. The coprocessor p_{SC} then returns the sealed timestamp

$$T(A) = E_{\text{tst}}(A, \mathbf{V}(A), S_{\text{tst}}(A, \mathbf{V}(A)))$$

(See Figure 5.)

The signature plays two roles here. First, it proves that this vector belongs to this event. Secondly, its presence *inside the plaintext* protects against a malicious process guessing the value of the vector, and verifying this guess using E_{tst}.

[2] This presentation assumes global schemes for all processes. In practice, giving each process its own key scheme adds flexibility and another level of security; Section 5.2 discusses these issues.

Comparing Timestamps When process p wants to compare events A and B, it sends $T(A)$ and $T(B)$ to p_{SC}. The coprocessor applies D_{tst} to extract the event names, vectors and signatures. If the signatures are valid, the coprocessor then compares $\mathbf{V}(A)$ and $\mathbf{V}(B)$, and reports the result: either "$A \longrightarrow B$," "$B \longrightarrow A$" or "$A \not\longleftrightarrow B$."

Sending Messages Suppose process p wants to execute a *send* event S, sending a message with text M to process q. Process p submits M and q to the secure coprocessor p_{SC}, which calculates the timestamp[3] $T(S)$, and returns the ciphertext

$$M' \quad = \quad E_{msg}(p, q, M, T(S), S_{msg}(p, q, M, T(S)))$$

(See Figure 6.) Process p then transmits the message.

A malicious process might still be able to suppress this message M. (For example, in Figure 1, *Bad Bob* could have his purchase order sealed, but only introduce it into the network if he receives an order from his client.) The secure coprocessor p_{SC} can protect against loss by requiring a signed acknowledgement from q_{SC}. If the acknowledgement does not arrive, p_{SC} can retransmit the message—perhaps incrementally, as part of other sealed packets. A malicious process can successfully suppress a message only by permanently partitioning itself from the network.

Receiving Messages Suppose a process p receives a ciphertext message M'. To read M', process p needs to send it to the secure coprocessor p_{SC}. The coprocessor applies D_{msg} to obtain the source and destination process, the plaintext M, the timestamp $T(S)$ of the *send* event, and the S_{msg} signature of this data. The coprocessor verifies that the S_{msg} signature is valid and that p is the intended destination process. The coprocessor then applies D_{tst} to the timestamp, checks its signature, and obtains the vector $\mathbf{V}(S)$. The coprocessor then performs the vector timestamp protocol: replacing its current vector V_p with the entry-wise maximum of V_p and $\mathbf{V}(S)$. Finally, p_{SC} returns to p the name of the source process, the plaintext M, and (optionally) the timestamp $T(S)$.

[3] Since messages are tagged with a signature before encrypting, using the unsealed timestamp $\mathbf{V}(S)$ would suffice here.

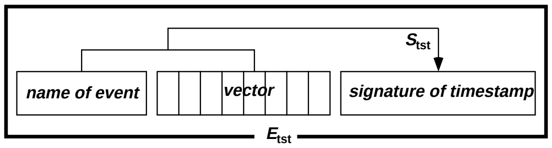

Figure 5: A sealed timestamp consists of the encryption of three items: the name of an event, its timestamp vector, and a signature on this pair. The signature certifies that this vector belongs to this event, and also protects against guessing the plaintext: verifying a guessed vector requires guessing the correct signature.

5 Discussion

5.1 Results

We make some preliminary observations.

The coprocessors carry out the vector timestamp protocol. This follows directly from the description.

Only secure coprocessors can unseal messages and timestamps. A process may be able to guess some or all of the entries of a given timestamp vector. If timestamps were merely vectors encrypted with a public key, then a process could guess a possible vector, encrypt the guess, and compare the result to the ciphertext. However, in our scheme, timestamps are the encryption of a vector along with a signature of that vector. Without knowing the signature function, a process cannot verify that V is the vector in the timestamp $E_{tst}(A, V, S_{tst}(A, V))$. Timestamps are truly sealed.

Similarly, with high probability a process cannot decrypt an encrypted message by making some lucky guesses, since that would require breaking the message signature S_{msg}.

Only the secure coprocessor at the source process may seal messages. Messages arriving at an honest process will be routed to the secure coprocessor, which will ignore messages that do not include both a valid timestamp and a valid signature on the message and the timestamp together.

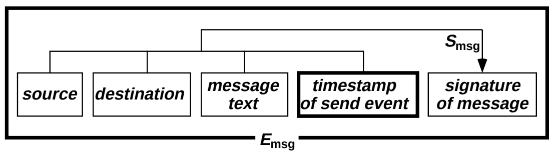

Figure 6: The message ciphertext encrypts the message information (source and destination processes, message text), along with the sealed timestamp of the *send* and a signature of these values.

Only the secure coprocessor at the intended destination process may unseal a message. Sealed messages must be decrypted to be intelligible. The receiving process must consult its secure coprocessor, since the encrypted message includes the name of the intended destination process. (However, a malicious process can receive and discard an encrypted message without consulting its coprocessor. Section 5.2 considers this.)

Together, these assertions imply the following result. (The proof can be found in [30].)

Theorem 2 *Sealed Vector clocks guarantee:*

- *If a clock reports "$A \longrightarrow B$" then $A \longrightarrow B$.*
- *If $A \longrightarrow B$ along a path where each message edge touches an honest process, then clocks will report "$A \longrightarrow B$."*
- *If $A \longrightarrow B$ along* any *path and there are no covert channels (i.e., malicious processes cannot communicate without using the sealed message protocol), then clocks will report "$A \longrightarrow B$."*

This protocol offers security advantages over prior work.

- **Complete Results** If a clock reports "$A \longrightarrow B$," then $A \longrightarrow B$. If a clock reports "$A \not\longleftrightarrow B$" (and malicious processes cannot communicate using covert channels) then $A \not\longleftrightarrow B$.
- **No Spoofing** Even with covert channels, a malicious process cannot deny having received a message from an honest process.
- **Privacy** The private information shared in timestamps is confined to the secure coprocessors.
- **Wider Application** The Sealed Vector Timestamp protocol does not require that the partial order directly arise from information flow.

In particular, Sealed Vectors protect against *all* the attacks catalogued in Section 3, and provide secure clocks for scenarios such as the partial order arising after rollback with modified replay. (See [30] for an example.)

Sealed Vectors also improve on Signed Vectors in terms of scalability: the number of decryptions required on incoming messages decreases from linear to constant.

5.2 Assumptions

This paper has made several implicit assumptions open to challenge. We discuss these challenges.

No Covert Channels Precedence corresponds to paths through the POT graph. The Sealed Vector protocol prevents a single malicious process from masking its presence in such paths. However, if malicious processes can communicate without using official (that is, coprocessor-sealed) messages, then they can cooperatively hide their presence in paths—since communication outside of the coprocessors is invisible to the clocks.

One approach to this problem is to make such communication very difficult: for example, by having the secure coprocessors handle net traffic (and perhaps snoop on Ethernet packets), malicious processes would be forced to communicate outside the network.

Covert communication is also possible using *in-band signaling*, since it may be possible to extract information from sealed messages without consulting secure coprocessors. For example, a malicious process might draw conclusions from the existence of the message, the length of the message (real encryption usually breaks long text into blocks and encrypts each block separately) or the frequency of multiple messages.

Security of Coprocessors The protocol depends on the physical security of the coprocessors. In practice, secure coprocessors are extremely difficult to penetrate. However, as with any security mechanism (physical or computational), it may be possible to compromise the system if the attacker is willing to pay tremendous amounts of money. (For a detailed analysis of the cost, see [38].) What do we do if the exception case occurs—if a coprocessor is compromised? One way to limit the damage is to use separate S_{msg}, S_{tst} and E_{msg} functions for each process. This technique prevents a compromised coprocessor from impersonating someone else or performing message decryption for someone else. Using separate E_{tst} functions prevents the compromised coprocessor from doing comparisons for someone else, but requires re-encrypting forwarded timestamps. (Section 5.3 considers some further defenses.)

Validity of Keys Giving each coprocessor its own keys raises the issue of key management: a new coprocessor must somehow announce its public keys. A straightforward technique to prevent dishonest processes from impersonating a "new coprocessor" is to have new coprocessors obtain certificates, signed by a universally trusted agent, listing their identity and public keys.

5.3 Future Work

Limiting Penetration Damage What can we do if the integrity of a coprocessor is compromised? Penetration exposes any data that a coprocessor has saved. However, an uncompromised coprocessor can securely *forget* data. This observation suggests an alternative *Give-and-Forget* timestamping scheme. Suppose process p at event S sends a message to process q, who receives it at event R. Process p generates a key pair $K_{1,S}$, $K_{2,S}$. Process p signs a certificate asserting that $K_{2,S}$ is its public key for event S, and sends this certificate along with the private key $K_{1,S}$ to process q with the message. Process q uses the private key $K_{1,S}$ to encrypt an identifier for R and then *erases the key.* Process q then has a universally verifiable certificate that it knew about S when R occurred. However, examining this certificate allows no one—not even process q—to forge a new certificate of knowledge of S without the cooperation of process p.

This technique allows a secure coprocessor to generate *proof-of-timestamp* certificates showing the last message received from each uncompromised process. Should the coprocessor later be compromised, it cannot produce new certificates for these messages. To prevent a compromised coprocessor from rolling back timestamp entries, we can require all coprocessors to use these proof-of-timestamp certificates to prove the validity of each entry in their timestamp vectors.

Other approaches for pre-compromised coprocessors to limit the forging power of their compromised versions include the *Distributed Trust* and *Digital Timestamping* techniques of [3, 13], as well using data on acknowledgement packets.

Improving Performance A performance problem with vector clocks results from size: timestamps have n entries; comparing timestamps requires n comparisons. Charron-Bost's result [6] that partial order timestamps must be linear suggests two approaches to improving performance: implementing vector clocks more carefully (to reduce the actual data transmitted), and trading timestamp *size* for comparison *time.*

Singhal and Kshemkalyani [28] present a vector clock implementation where processes refrain from transmitting redundant data in vectors. Integrating this technique with Sealed Vectors would yield increased efficiency.

Another interesting approach would be to give processes more latitude in choosing which entries to transmit and which to withhold. Some entries in timestamp vectors might be marked with flags indicating that that value is merely a lower bound. This lower bound may suffice for many comparisons; if it doesn't, a secure coprocessor would need to consult other secure coprocessors to obtain the missing data. It would be interesting to develop good heuristics for deciding which entries to withhold and for determining when the expense of a "miss" outweighs the benefits of withholding.

Yet another technique (e.g., [1]) is to use vector clocks to track a coarser partial order—thus trading timestamp size for false positives in precedence detection. However, adapting these techniques (or the linear timestamping techniques of [3, 13]) creates the problem of proving the *absence* of a precedence path. Developing a hierarchical approach—to indicate the most "likely" precedence path, and then verify its correctness—is one path of future research.

General Confinement Models Another area for exploration is the use of more general confinement models. Coprocessor sealing provides control over the information a timestamp provides to a process. This control may provide more benefits than just suppressing vector entries—in particular, it may allow for anonymous or hidden causality [12].

Secure Distributed Applications We have used secure clocks for partial order time (and for more general temporal structures) to develop secure protocols for problems such as immediate ordered service, distributed snapshots, and optimistic rollback recovery. ([30, 31] contain more details.) We are continuing to explore new applications using our secure time framework.

Acknowledgments

We are grateful to Gene Cooperman, Dave Johnson, "Bad" Bob Wheeler, and Bennet Yee for helpful comments.

References

[1] M. Ahuja, T. Carlson, A. Gahlot, and D. Shands, *Timestamping Events for Inferring "Affects" Relation and Potential Causality*, Computer and Information Science Technical Report OSU-CISRC-5/91-TR13, Ohio State University, May 1991.

[2] W. Alexi, B. Chor, O. Goldreich, and C.P. Schnorr, "RSA and Rabin Functions: Certain Parts are as Hard as the Whole," *SIAM Journal on Computing*, Vol. 17, pp. 194-209, 1988.

[3] D. Bayer, S. Haber, and W.S. Stornetta, "Improving the Efficiency and Reliability of Digital Time-Stamping," *Sequences II: Methods in Communication, Security, and Computer Science*, Springer Verlag, 1993.

[4] M. Blum and S. Goldwasser, "An Efficient Probabilistic Public-Key Encryption Scheme which Hides All Partial Information," *Advances in Cryptology: Proceedings of Crytpo 84.* Springer Verlag LNCS 196.

[5] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems.* Vol. 3, pp. 63-75, February 1985.

[6] B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters*, Vol. 39, pp. 11-16, July 1991.

[7] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, Vol. IT-22, pp. 644-654, November 1976.

[8] C.J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," *11th Australian Computer Science Conference,* pp. 56-67, February 1988.

[9] C.J. Fidge, "Logical Time in Distributed Computing Systems," *IEEE Computer*, Vol. 24, pp. 28-33, August 1991.

[10] O. Goldreich, *Foundations of Cryptology*, Computer Science Department, Technion, 1989.

[11] S. Goldwasser and S. Micali, "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information," *14th ACM Symposium on Theory of Computing*, 1982.

[12] I.G. Greif, *Semantics of Communicating Parallel Processes*, Ph.D. thesis, Massachusetts Institute of Technology, 1975.

[13] S. Haber and W.S. Stornetta, "How to Time-Stamp a Digital Document," *Journal of Cryptology*, Vol. 3, pp. 99-111, 1991.

[14] D.B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, Vol. 11, pp. 462-491, September 1990.

[15] P. Kearns and B. Koodalattupuram, "Immediate Ordered Service in Distributed Systems," *9th Symposium on Reliable Distributed Systems*, 1989.

[16] A.D. Kshemkalyani and M. Singhal, *Characterization of Distributed Deadlocks*, Computer and Information Science Technical Report OSU-CISRC-6/90-TR15, Ohio State University. June 1990.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, pp. 558-565, July 1978.

[18] R. Lipton, *How to Cheat at Mental Poker*, personal communication, 1981.

[19] K. Marzullo and G. Neiger, "Detection of Global State Predicates," in Toueg, Spirakis and Kirousis (ed.), *5th International Workshop on Distributed Algorithms (WDAG-91),* Springer-Verlag LNCS 579, 1991.

[20] F. Mattern, "Algorithms for Distributed Termination Detection," *Distributed Computing*, Vol. 2, pp. 161-175, 1987.

[21] F. Mattern, "Virtual Time and Global States of Distributed Systems," in Cosnard, et al, ed., *Parallel and Distributed Algorithms,* Amsterdam: North-Holland, 1989, 215-226.

[22] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, Vol. 18, pp. 423-434, August 1993.

[23] S.L. Peterson and P. Kearns, "Rollback Based on Vector Time," *12th Symposium on Reliable Distributed Systems*, October 1993.

[24] V.R. Pratt, "Modeling Concurrency with Partial Orders," *International Journal of Parallel Programming*, Vol. 15, pp. 33-71, 1986.

[25] M. Rabin, *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*, Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, January 1979.

[26] M. Reiter and L. Gong, "Preventing Denial and Forgery of Causal Relationships in Distributed Systems," *1993 IEEE Symposium on Research in Security and Privacy.*

[27] R. Rivest, A. Shamir and L. Adlemann. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, Vol. 21, pp. 120-126, 1978.

[28] M. Singhal and A.D. Kshemkalyani, *An Efficient Implementation of Vector Clocks*, Computer and Information Science Technical Report OSU-CISRC-11/90-TR34, Ohio State University, November 1990.

[29] S.W. Smith, *A Theory of Distributed Time*, Computer Science Technical Report CMU-CS-93-231, Carnegie Mellon University, December 1993.

[30] S.W. Smith, *Secure Distributed Time for Secure Distributed Protocols*, Ph.D. thesis, Computer Science Technical Report CMU-CS-94-177, Carnegie Mellon University, 1994.

[31] S.W. Smith, D.B. Johnson, and J.D. Tygar, *Asynchronous Optimistic Rollback Recovery Using Secure Distributed Time*, Computer Science Technical Report CMU-CS-94-130, Carnegie Mellon University, March 1994.

[32] S.W. Smith and J.D. Tygar, *Signed Vector Timestamps: A Secure Protocol for Partial Order Time*, Computer Science Technical Report CMU-CS-93-116, Carnegie Mellon University, October 1991 (version of February 1993).

[33] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, pp. 204-226, August 1985.

[34] Y.C. Tay and W.T. Loke, *A Theory for Deadlocks*, Computer Science Technical Report CS-TR-344-91, Princeton University. August 1991.

[35] J.D. Tygar and B.S. Yee, "Dyad: A System for Using Physically Secure Coprocessors," *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, April 1993. (A preliminary version is available as Computer Science Technical Report CMU-CS-91-140R, Carnegie Mellon University.)

[36] L. Van Valkenburg, personal communication, July 12, 1994.

[37] S.H. Weingart, "Physical Security for the μABYSS System," *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, 1987.

[38] S.H. Weingart, *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses*, IBM, internal use only, March 1991.

[39] S.R. White, S.H. Weingart, W.C. Arnold, and E.R. Palmer, *Introduction to the Citadel Architecture: Security in Physically Exposed Environments*, Technical Report, Distributed Security Systems Group, IBM Thomas J. Watson Research Center, March 1991.

[40] Z. Yang and T.A. Marsland, *Global States and Time in Distributed Systems*, IEEE Computer Science Press, 1994.

[41] B.S. Yee, *Using Secure Coprocessors*, Ph.D. thesis, Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University, May 1994.