Security and Privacy for Partial Order Time

S.W. Smith J.D. Tygar

April 1994

CMU-CS-94-135

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

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* protocol provides a partial order time service with some security: no one can forge dependence on an honest process. However, that protocol still permits some forgery of dependence, permits all denial of precedence, and leaks 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 causal paths even in the presense of malicious processes, and protects against some privacy risks as well. By solving these previously open security problems, our new protocol provides a foundation for incorporating security and privacy into distributed application protocols based on partial order time.

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.

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

1. Introduction

Motivation Partial order time is central to solving application problems in asynchronous distributed systems. Since protocols for these applications require examining an underlying partial order, explicitly providing a partial order time service simplifies and clarifies the task of protocol design.

However, while the passage of real time can be determined from examining an independent physical device, partial order time cannot be maintained in isolation. Tracking partial order time requires collecting and sharing information. Thus dealing with partial order time—even implicitly—exposes protocols to security risks. Is the information a process receives correct? Is the information a process shares being used for dishonest purposes?

Encapsulating a system's dealings with partial order time into a single time service provides another benefit: a single arena in which to examine and resolve these security issues.

Previous Work In earlier work [SmTy91], we recognized the central role of partial order clocks, catalogued some of the security and privacy risks, and presented the *Signed Vector Timestamp* protocol,[1] which protects against some of these risks. While this protocol prevents prevents dishonest processes from forging causal dependence on honest events, it suffers from some drawbacks:

- The Signed Vector protocol cannot guarantee detection of causal paths touching dishonest processes. Consequently, Signed Vectors cannot be used to build secure protocols for problems such as *distributed snapshots* requiring accurate detection of non-precedence.
- The Signed Vector protocol leaks private information, since vector entries are publically readable.
- The Signed Vector protocol requires that the temporal relation being tracked express all paths of information flow; thus the protocol does not extend to more general relations.

This Paper In this paper, we use new developments in inexpensive tamper-proof hardware to build the *Sealed Vector* protocol, which provides stronger security and privacy protection than any previous protocol. In particular, our new protocol prevents dishonest processes from forging causal dependence on *any* events, and (if malicious processes cannot communicate covertly) prevents dishonest processes from denying causal dependence. (Even with covert communciation, Sealed Vectors provide some protection against denying causal dependence.) Thus this work solves previously open problems [ReGo93].

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

[1] See also [ReGo93].

2. Partial Order Clocks

Time organizes experience. Traditionally we use real time to organize experience into a linear sequence of events. However, in asynchronous distributed systems, linear time is often inappropriate [La78, Pr86]. Application problems decompose into questions about more general temporal relations; clocks for these relations would provide building blocks for application solutions. These applications include the following:

- *snapshots and global states* [MaNe91, MaSa91, Ma93, Sm94]
- *deadlock detection* [KsSi90, Ma87, TaLo91]
- *immediate ordered service* [KeKo89]
- *optimistic rollback recovery* [Jo89, JoZw90, PeKe93, SJT94, StYe85]

2.1. Partial Order Time

Partial order time (POT) is the major alternative time model to global sequential time. Suppose our system consists of a collection of n processes, each of which experiences a linear sequence of events. Suppose further that these processes are asynchronous: real time clocks do not exist, and the duration between consecutive events at a process is unpredictable. The processes communicate by passing messages, which arrive either once after an unpredictable positive delay, or never.

We can represent the behavior of this system as a directed graph: construct a node for each event, and draw edges connecting consecutive events at each process and from the *send* of a message to its *receive*. We call this the POT graph of the computation; its transitive closure $\overline{\text{POT}}$ determines a partial order on the 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.

2.2. Clocks

Clocks for partial order time should be devices that allow processes to determine the precedence (or concurrency) of events during a computation. That is, at some event C, process p should be able to send the query $\mathsf{Precedes}(A, B)$ to its clock to learn whether or not event A precedes event B in the partial order.

This sketch raises some issues. How do processes refer to events? At event C, what events A and B should process p be able to ask about? Will the clock at process p have enough information to answer? Will the answer ever change?

To answer the first questions, we assume event names include the process at which they occur and their index in the local sequence there. Further, we assume that a process p is allowed to ask about anything it knows about and that the POT model captures the flow of knowledge. At event C, the clock at process p better be able to handle queries for all A, B such that $A \rightleftharpoons C$ and $B \rightleftharpoons C$.

The POT model possesses a convenient monotonicity that answers the final question. When an event occurs in a computation, it is added to the POT graph along with some incoming edges, but no more incoming edges are added after that point. Thus once events A and B exist, either $A \longrightarrow B$ always or $A \not\longrightarrow B$ always.

2.3. Vector Timestamps

The assumption that POT edges are the only channels for information flow, along with the monotonicity of POT graphs, suggest a *timestamp* approach: 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.

A well-known timestamp method for partial orders is the *Vector Clock* approach [StYe85, Ma87, Fi88]. Each process maintains a local event counter, and timestamps each event A with a vector $\mathbf{V}(A)$ consisting of the local index of the maximal event at each process that precedes or equals A. That is, the process q entry of $\mathbf{V}(A)$ is the q-index of the event B at q such that:

- $B \rightleftharpoons A$
- For any C at q, if $C \rightleftharpoons A$ then $C \rightleftharpoons B$

This definition fails when no event at some q precedes an event A. To handle this case, we adopt the convention of that a global event $\perp$ is the initial event at each process.

The linear ordering of events at a process suggests a natural ordering on vector timestamps: for vectors V and W, we say that V precedes W

$$V \prec W$$

when for each process p, the process p entry of V precedes or equals the p entry of W in the linear order at p, but for some process p, this inequality is strict.

Establishing that vector timestamps function as clocks follows directly.

Theorem 1 For events A and B, $\mathbf{V}(A) \prec \mathbf{V}(B) \iff A \longrightarrow B$.

Implementing Vector Timestamps Implementation of vector timestamps also follows easily. Each process p maintains a vector V for its most recent event. When a new event occurs, process p

increments the p entry of V to obtain the new current value V'. If this new event is a *send*, process p appends the new value V' to the message. If this new event is a *receive*, process p strips off the timestamp W from the message and replaces V' with the entry-wise maximum of V' and W.

3. Security and Privacy Attacks

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 sharing and trusting creates opportunities for Byzantine (malicious) processes to manipulate the clock protocols, and consequently to manipulate application protocols built on these clock protocols.

We sketch four such attacks on vector clocks.

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. Simple sanity checks fail to combat this problem: 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 honest victim may then be mistakenly identified as corrupt by the next honest process it talks to.

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 options on a public network. Figure 1 shows how this technique permits the crime of *options frontrunning*, which occurs in the (physical) Chicago commodities exchange, where brokers can trade both for themselves and their clients. 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. This profit provides incentive for a dishonest broker, upon receiving a client request, 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 merely by using an old vector on his purchase request.[2]

Malicious Postdating Malicious proceses can selectively *inflate* vector entries, and thus fool honest processes into thinking events happened later than they really did. Figure 2 shows how this technique 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.)

[2]Such time forgery is even easier in the physical environment of the Chicago exchange; currently, the only defense the FBI has against options frontrunning is placing undercover agents in the pit to look for unusually lucky brokers.

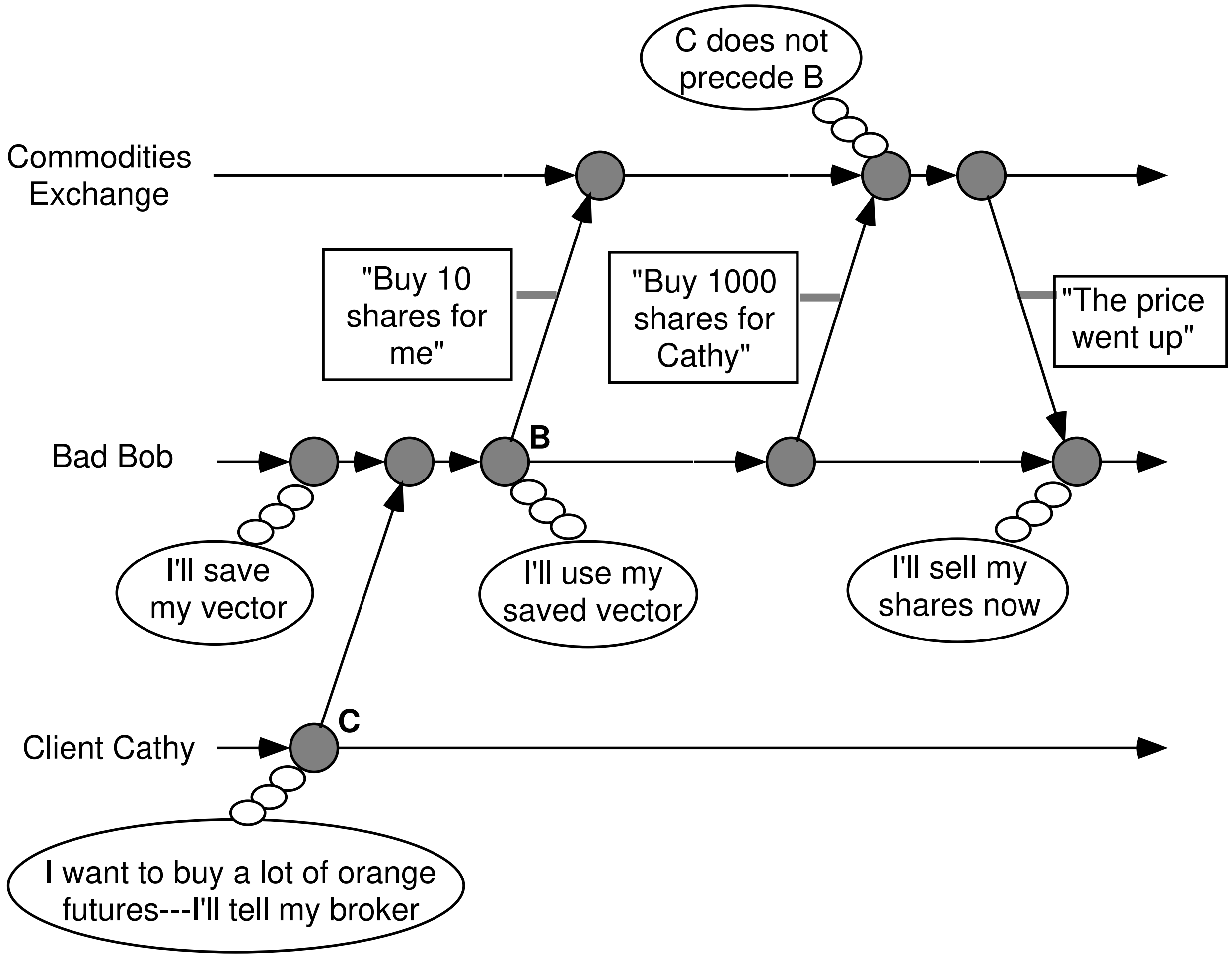

Figure 1 Malicious processes can selectively backdate events. Here, *Bob* commits the crime of *options frontrunning* by making his own puchase appear not to follow his client's request.

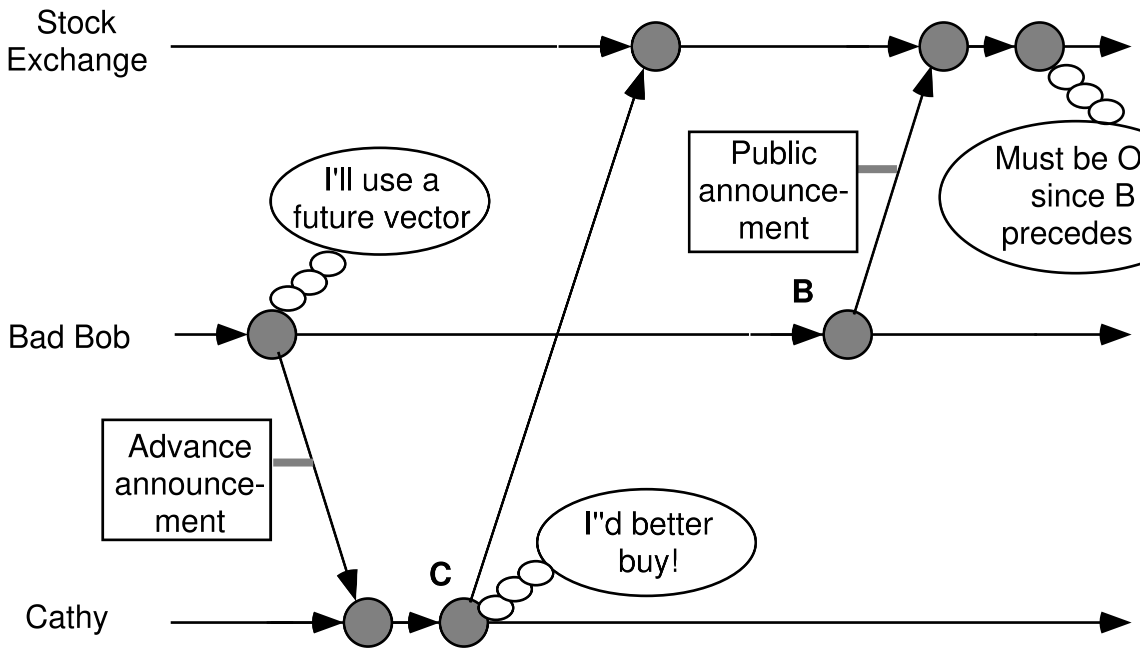

Figure 2 Malicious processes can selectively postdate events. Here, *Bob* leaks an advance copy of his public announcement to *Cathy* in such a way that allows her to act on the data first, without appearing to have had a headstart.

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 leads to compromising the identify of whistleblowers. The changes in subsequent timestamp vectors sent from *Alice* to *Bob* betrays to whom *Alice* has been talking in the interim.

4. Defending Against These Attacks

Ideally, the clock at an honest process should report "$A \longrightarrow B$" iff $A \longrightarrow B$, while confining private information, no matter what actions malicious processes take. This standard provides a scheme for evaluating clock protocols: against decreasing amounts of honesty, how well do they perform?

Many application protocols use forms of partial order time and vector clocks. If a clock system meets this ideal, then installing these clocks into these higher-level application protocols will transparently protect these protocols against the security and privacy attacks of Section 3.

4.1. Previous Work

If all proceses are honest, then the process p entry in anyone's vector timestamp orignated at process p. Our *Signed Vector Timestamp* protocol [SmTy91] builds on this observation by requiring each process to digitally "sign" its entries in outgoing timstamp vectors. This scheme prevents malicious processes from advancing the vector entries belonging to honest processes. If the process where an event A occurs is honest, and our time model expresses all paths of information flow, then possession of a signed entry for event A is proof of dependence on A. Thus Signed Vectors provide $A \longrightarrow B$ whenever an honest clock reports "$A \longrightarrow B$"; if everyone along a causal path from A to B is honest, the protocol also provides the converse: $A \longrightarrow B \implies$ "$A \longrightarrow B$."

However, since malicious processes can use old values in the entries for honest processes (all that's required is a signature-value pair), any causal path touching a malicious process may not necessarily be detected. Consequently, Signed Vectors still permit *Malicious Backdating*, *Malicious Postdating*, and *Compromised Privacy*. Further, the inability of Signed Vectors to reliably report non-precedence makes it difficult to transfer their security to protocols for higher-level problems that require detection of non-precedence. The consequences of this inability can range from inefficiency (in *optimistic rollback recovery*, processes may mistakenly believe they depend on failed states) to complete incorrectness (in *global state* protocols, processes may make incorrect decisions regarding "concurrent" events).

Reiter and Gong [ReGo93] also present the Signed Vector protocol, and propose three additional protocols for the special case of a process sorting the *send* events of two messages it has received. Their *Piggybacking* protocol generalizes the vector timestamp protocol by passing around a condensed version of the history of an event. It provides the same properties as Signed Vectors, and

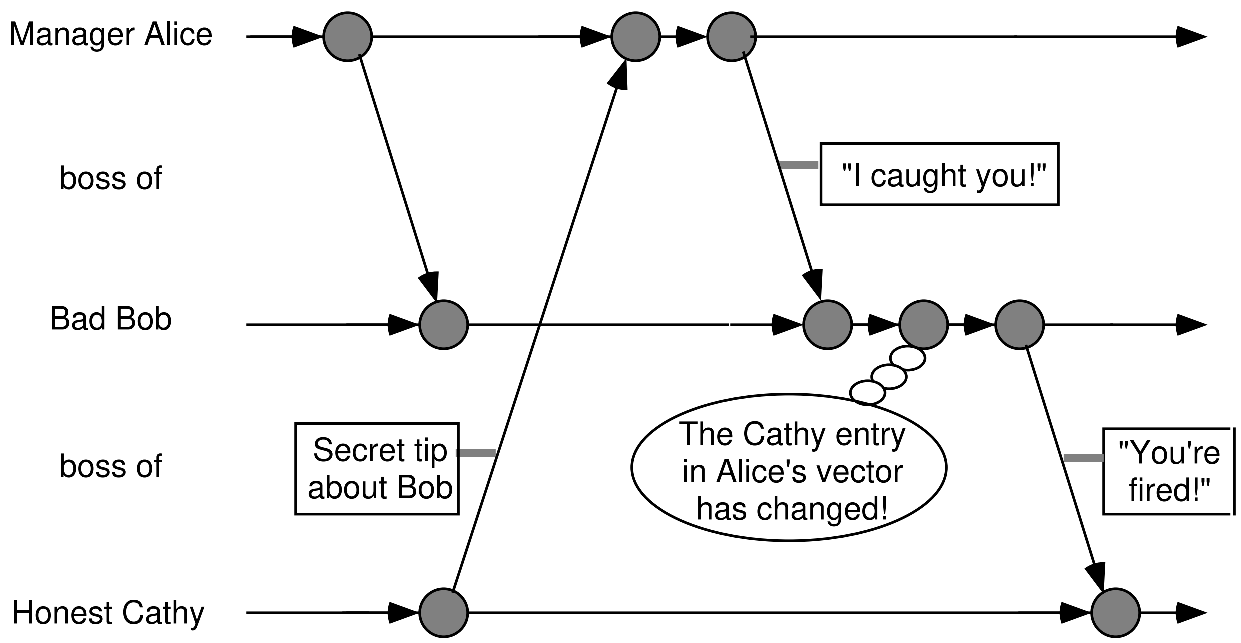

Figure 3 Malicious processes can exploit vector data for illicit purposes. Here, *Bob* uses the timestamp vectors from *Alice* to learn the identity of whistle-blower *Cathy*.

provides some additional power in the case of complete dishonesty. The Piggbybacking protocol extends to handle the general case of sorting arbitrary events.

Their other two protocols alter the order in which messages are received. Thus, these protocols "solve" 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.

Our New Results The challenge remains of building a protocol that can accurately report "$A \longrightarrow B$" or "$A \not\longrightarrow B$" even when A's process is corrupt, and that can do so without leaking information. Our new *Sealed Vector Timestamp* protocol solves these open problems, satisfies the ideal (assuming no covert channels), and protects privacy of vector entries as well. Further, this protocol will extend to more general time models as well—since we do not really need to have our time model express all paths of information flow. Figure 4 summarizes these protocols.

4.2. The Sealed Vector Protocol

This section presents our new protocol. Section 4.2.1 discusses the physical tools; Section 4.2.2 presents an overview of the protocol; and Section 4.2.3 reviews the cryptographic tools. Section 4.2.4 presents the details of protocol operations; Section 4.2.5 presents the security results; and Section 4.2.6 discusses some drawbacks.

who's honest?	"$A \longrightarrow B$" $\Longrightarrow$ $A \longrightarrow B$	"$A \longrightarrow B$" $\Longleftarrow$ $A \longrightarrow B$	"$A \longrightarrow B$" $\Longleftrightarrow$ $A \longrightarrow B$	privacy of entries
A, B and path between them	Signed, PB, Sealed	Signed, PB, Sealed	Signed, PB, Sealed	Sealed
only A	Signed, PB, Sealed	Sealed	Sealed	Sealed
no one (but querying process)	Sealed	Sealed	Sealed	Sealed

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

4.2.1. Secure Coprocessors

Our new protocol rests on the the technology of *secure coprocessors* [TyYe93, Yee94]: inexpensive physically secure devices with a limited amount of computational power, ROM, and non-volatile RAM. A process 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 resetting of RAM and CPU registers to null values.

A key notion here is that a secure coprocessor only possesses a *limited* amount of power. We cannot secure an entire workstation—even if we could, we could not secure the user!

Secure coprocessors are being rapidly deployed; there are now several commercial secure coprocessor procucts available from IBM (μABYSS [Wein87], Citadel [WWAP91]) and other vendors. Transforming this small amount of physical security into a larger secure protocol raises some subtle issues. For example, malicious processes might attempt to bypass their coprocessors, or to attack the communication lines between. (See [TyYe93, Yee94] for information on how to protect against these attacks.)

4.2.2. Overview

The main idea for Sealed Vectors is to grant each process a secure coprocessor that creates timestamp vectors, and *seals* them so that processes cannot read them. Processes can store and pass around timestamps, but need to query a secure coprocessor in order to compare them.

This scheme requires a number of properties:

- No one (except a secure coprocessor) can obtain information about the contents of any vector entry from an encrypted timestamp, even if they know the other entries.
- All processes must route a message from an incoming process through the secure coprocessor.
- All processes must route a message to an outgoing process through the secure coprocessor—and once routed, must not be able to forever suppress this message.
- A secure coprocessor must be able to verify that a timestamp given to it was produced by another secure coprocessor.

Thinking about how processes will use timestamps raises another question. Process p at event C may want to determine whether two events A and B (with $A \Longrightarrow C$ and $B \Longrightarrow C$) satisfy $A \longrightarrow B$. Thus process p needs to know the timestamps of A and B, so it can feed them to its secure coprocessor. Presumably process p trusts itself, so if either of these events occurred at p, then p knows its timestamp. However, suppose A occurs at process $q \neq p$. According to the timestamp clock model, process p obtains the A timestamp the same way it obtains the name of the A—through information passed along POT graph edges. How can process p be sure that somewhere along this route, a malicious process did not substitute a different timestamp for A?

This problem illustrates the need for another property.

- It must be possible to verify that a given timestamp belongs to a given event.

4.2.3. Cryptographic Tools

We build a timestamp scheme meeting this description using two common cryptographic tools.

Digital Signatures A digital signature scheme is a function S from a value space to a signature space such that:

- Given a value v and a signature s, anyone can tell whether or not s is a valid signature of v: whether or not the signature function S maps s to v.
- However, with high probability, no one except a privileged agent can take a set of value-signature pairs and efficiently produce a pair not in this set.

Bit-Secure Encryption A public key encryption scheme consists of a function E (from the value space to the cipherpace) and a function D (from the cipherspace to the value space) such that:

- For any value v, anyone can calculate $E(v)$.
- For any value v, $D(E(v)) = v$.

Ordinary public key encryption requires only that inversion of E is difficult (without the privilege of knowing D). We require an additional level of *security* [Gold89]. From a given ciphertext, a malicious process should gain no information about the plaintext that it did not know *a priori*. Some popular cryptosystems (like [RSA78] and [Ra79]) are known to leak number-theoretic properties of the plaintexts and thus fail to meet this condition [ACGS88, Li81]. For the Sealed Vector protocol to attain its full security potential, it should be implemented using strong cryptosystems such as [GoMi82] or [BlGo84].

Our scheme requires encryption and signatures for messages and timestamps. For the initial presentation of the protocol, we assume a a single global scheme for messages (E_{msg}, D_{msg} and S_{msg}), and another for timestamps (E_{tst}, D_{tst} and S_{tst}).

(In practice, giving each process its own key scheme adds flexibility and another level of security; Section 4.2.6 discusses these issues.)

4.2.4. Details

By definition, the functions E_{msg} and E_{tst} are public. Each process p has a secure coprocessor p_{SC}. The coprocessor p_{SC} knows that it belongs to process p, and has the ability to calculate D_{msg}, D_{tst}, S_{msg}, and S_{tst}. The coprocessor p_{SC} also maintains the timestamp vector V_p (and thus maintains an official "event count" for process p).

Obtaining Timestamps Suppose process p wants to obtain a timestamp for its current event A. It submits the request to p_{SC}, which increments the p entry of V_p, so it then equals $\mathbf{V}(A)$, and returns the sealed timestamp

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

Figure 5 illustrates the structure of sealed timestamps.

The signature plays two roles here. First, it proves that this vector really belongs to this event. Secondly, its presence *inside the plaintext* protects against plaintext attacks. A malicious process must break the signature function in order to to verify that a particular encrypted timestamp contains a particular vector value. Section 4.2.5 discusses this issue further.

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

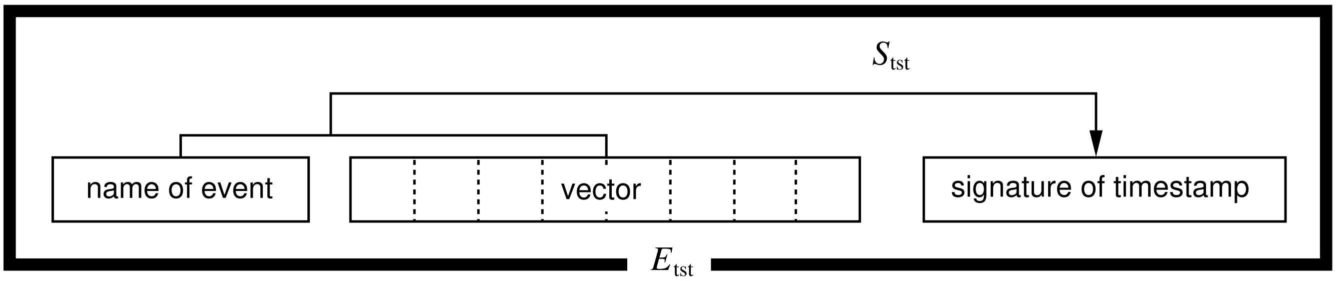

Figure 5 A sealed timestamp consists of the encryption of the name of an event, its timestamp vector, and a signature on this pair. The signature certifies that this vector belongs to this event, and also protects against guessing the plaintext: verifying a guessed vector requires guessing the correct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

calculates the timestamp $T(S)$, and returns the ciphertext

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

Figure 6 illustrates the structure of this ciphertext. Process p then transmits the message (alternatively, the coprocessor itself could send the message directly.)

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

A malicious process—either p, or anyone along the network—could still suppress this message M. (For example, in Figure 1, *Bob* could have his purchase order sealed, but only introduce it into the network if he receives an order from his client.) We can protect against such deliberate loss by having p_{SC} expect 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 suppress a message only by forever separating 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 this signature is valid and that p is the 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 maxima of V_p and $\mathbf{V}(S)$. Finally, the coprocessor returns the source process name, the plaintext M, and (optionally) the timestamp $\mathbf{V}(S)$ to the process p.

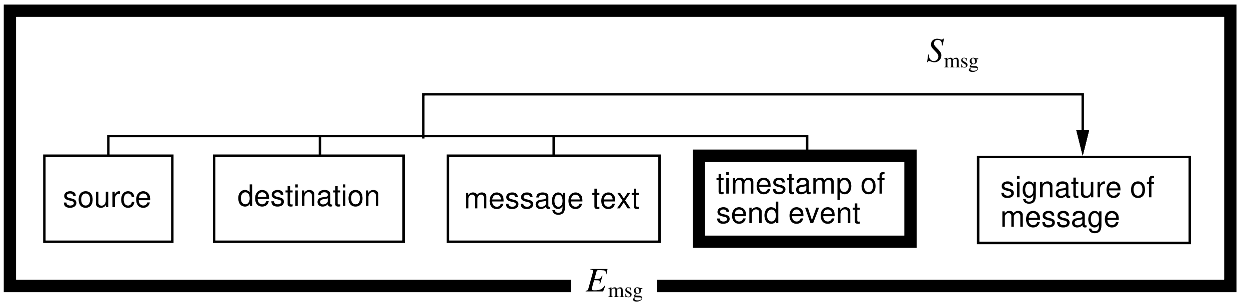

Figure 6 The message ciphertext consists of encrypting the message information (source and destination processes, message text), along with the sealed timestamp of the *send* and a signature of these values. The presence of the signature makes the encrypted package invulnerable to plaintext attacks, so the receiver must consult a secure coprocessor. Since the name of the receiving process is included in the package, decryption will only be performed by the secure coprocessor of this process.

4.2.5. Results

Establishing the security and correctness of the sealed vector protocol requires some observations.

The coprocessors carry out the vector timestamp protocol This follows directly from the description. (However, the only events that officially occur in the partial order time framework are those for which a process requests timestamps.)

Messages and timestamps are invulnerable to plaintext attacks 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 malicious process could use its knowledge about possible vectors to obtain actual vector entries by guesssing a possible vector, encrypting the guess, and comparing the result to the ciphertext. However, in our scheme, timestamps are the encryption of a vector along with a signature of that vector: thus even if a process knows the value of a vector V, it cannot verify that V is indeed the vector encoded in the timestamp $E_{\text{tst}}(A, V, S_{\text{tst}}(A, V))$. Timestamps are truly sealed.

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

Coprocessors must examine messages to honest processes Sending a message to an honest process requires obtaining both a valid timestamp and a valid signature on the message and the timestamp together.

Coprocessors must examine messages from honest processes Receiving a message from an honest process requires decryption in order to understand it. Since (by above) encrypted messages are invulnerable to plaintext attacks, a process must consult a coprocessor; since the encrypted message includes the name of the receiving process, a process must consult its own coprocessor. (However, a malicious process can receive and discard an encrypted message without consulting its coprocessor. Section 4.2.6 considers this avenue.)

Together, these assertions imply the following result.

> **Theorem 2** Suppose all messages to or from honest processes are routed through through secure coprocessors, the encryption and signature functions are not breakable, and the secure coprocessors are not compromised. Then Sealed Vectors guarantee the following:
>
> - If a clock reports "$A \longrightarrow B$" then $A \longrightarrow B$.
> - If event A precedes event 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 *malicious processes cannot communicate without using the sealed message protocol*, then clocks will report "$A \longrightarrow B$."

(Since secure coprocessors handle clock queries, all clocks are honest.)

We discuss some advantages of this protocol 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 pretend not to have received a message from an honest process.
- **Privacy** The private information shared in timestamps cannot be exploited for improper use, since it is not readable.

In particular, the Sealed Vector protocol protects against *all* the attacks catalogued in Section 3.

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

4.2.6. Drawbacks

Theorem 2 establishes the security properties of Sealed Vectors, but incorporates three hypotheses open to challenge. We discuss these challenges.

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 proceses would be forced to run a separate cable. Nevertheless, a coterie of covert communicators essentially becomes one process. How can two secure coprocessors tell they are attached with the same process? Alternatively, what precedence information can honest processes obtain despite covert channels? These remain research questions.

In-band Signaling The encryption tools prevent forging sealed messages: sending a message to an honest process requires consulting one's secure coprocessor. The protocol seals messages in order to force a malicious process to consult its coprocessor before extracting any information from an incoming message. However, it may be possible to extract information without such consultation—by using 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. One way to suppress this style of attack is to hide network activity; perhaps by connecting the network directly to the secure coprocessors or by more careful structuring of network packets.

Compromised Integrity 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 [Wein91].) 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 comparisions for someone else, but requires re-encrypting forwarded timestamps.

(Section 5.1 considers some further defenses.)

Key Management 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. Future Work

5.1. Limiting Potential Damage from Penetration

What can we do if the integrity of a coprocessor is compromised?

Give and Forget Penetration exposes any data that a coprocessor has saved. However, an uncompromised coprocessor can securely *forget* data. This observation suggests an alternative timestamping scheme. Suppose process p, at event S, sends a message to process q, who receives 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 ket 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 event process q—to forge a certificate of knowledge of S (without the cooperation of process p).

This technique allows a secure coprocessor to generate certificates showing the last message it received from each honest process. However, 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 coprocessors to use these certificates to prove the validity of their vector entries. (The transitivity of dependence makes this task somewhat non-trivial.)

Independently Tracking Progress Other approaches for pre-compromised coprocessors to limit the forging power of their compromised versions include the *distributed trust* and *digital timestamping* techniques of [BHS92, HaSt91], as well using data on acknowledgement packets.

5.2. Improving Performance

A serious performance problem with vector clocks is size: timestamps have n entries; comparing timestamps requires n comparisons. Reducing these bounds provides some grounds for future work.

Charron-Bost's result [ChBo91] that partial order timestamps must be linear suggests two approaches: implementing vector clocks more carefully (to reduce the actual data transmitted), and trading timestamp *size* for comparison *time*.

Efficient Vector Clocks Singhal and Kshemkalyani [SiKs90] present a vector clock implementation where processes refrain from transmitting redudnant data in vectors. Incorporating this technique with our 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 would be marked with flags indicating that that value is merely a lower bound. This lower bound may suffice for many comparisons; when or where it doesn't, a secure coprocessor needs to consult other secure coprocessors to obtain the missing data.

Developing good heuristics to use in deciding which entries to withhold and determining when the expense of a "miss" outweighs the benefits of withholding will provide the basis for some interesting experiments.

Centralized Vector Clocks Another approach is implementing the vector clock protocol in a more centralized fashion. For the extreme case, suppose we had a single trusted *logging site*. When a process receives a message, its secure coprocessor sends a note to the logging site indicating the sending process, the receiving process, and the local indices of the *send* and *receive* events. The logging site then has sufficient information to maintain the timestamp vectors for each process. We obtain constant size timestamp data on messages—at the price of doubling the number of messages, and having processes need to consult a remote site to perform comparisons.

This approach still requires coprocessor sealing in order to force a process not only to acknowledge receiving a message, but also to file a logging note. (This approach differs from the *Causality Server* protocol [ReGo93] in that messages are not routed through an an intermediary, but logged after the fact, that no FIFO nor secure channel assumptions are needed, and that the logging site protocol preserves the actual partial order, not just a consistent total order.)

Hierarchies of Vector Clocks Another technique (e.g., [ACGS91]) 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 (or the linear timestamping techniques of [BHS92, HaSt91]) creates the problem of proving the *absence* of a causal path. Developing a hierarchical approach—to indicate the most "likely" causal path, and then verify its correctness—is one path of future research.

5.3. Improving Security and Privacy

The Sealed Vector protocol improves on our earlier work [SmTy91] by preventing the processes from being fully conscious of all local computation. One direction for future work lies in exploring the potential of this technique. We briefly discuss two such areas.

Suppressing Traffic Analysis The Sealed Vector protocol prevents malicious processes from reading actual vector entries. However, malicious proceses may obtain private information by other means, such as analyzing network traffic. Since processes cannot read *any* packets sent by secure coprocessors, secure coprocessors can transmit random packets—that malicious processes cannot suppress without detection.

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.

References

[ACGS91] Ahuja, M., 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. May 1991.

[ACGS88] 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*, 17:194-209. 1988

[BHS92] 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.

[BlGo84] 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.

[ChBo91] B. Charron-Bost. "Concerning the Size of Logical Clocks in Distributed Systems." *Information Processing Letters.* 39:11-16. July 1991.

[Fi88] C.J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering." *11th Australian Computer Science Conference.* 56-67. February 1988.

[Fi91] C.J. Fidge. "Logical Time in Distributed Computing Systems." *IEEE Computer.* 24 (8):28-33. August 1991.

[Gold89] O. Goldreich. *Foundations of Cryptology.* Computer Science Department, Technion, 1989.

[GoMi82] 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.

[HaSt91] S. Haber and W.S. Stornetta. "How to Time-Stamp a Digital Document." *Journal of Cryptology*, 3: 99-111. 1991.

[Jo89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Ph.D. thesis, Rice University, 1989.

[JoZw90] D.B. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing." *Journal of Algorithms.* 11: 462-491. September 1990.

[KeKo89] P. Kearns and B. Koodalattupuram. "Immediate Ordered Service in Distributed Systems." *9th Symposium on Reliable Distributed Systems.* IEEE, 1989.

[KsSi90] A.D. Kshemkalyani and M. Singhal. *Characterization of Distributed Deadlocks.* Computer Science Technical Report TR OSU-CISRC-6/90-TR15, Ohio State University. June 1990.

[La78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*. 21: 558-565. July 1978.

[Li81] R. Lipton. *How to Cheat at Mental Poker.* Personal communication, 1981.

[MaNe91] 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.

[MaSa91] K. Marzullo and L. Sabel. "Using Consistent Subcuts for Detecting Stable Properties." In Toueg, Spirakis and Kirousis (ed.), *5th International Workshop on Distributed Algorithms (WDAG-91).* Springer-Verlag LNCS 579. 1991.

[Ma87] F. Mattern. "Algorithms for Distributed Termination Detection." *Distributed Computing.* 2: 161-175. 1987.

[Ma89] 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.

[Ma93] Mattern, F. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing.* 18: 423-434. August 1993.

[PeKe93] S.L. Peterson and P. Kearns. "Rollback Based on Vector Time." *12th Symposium on Reliable Distributed Systems.* IEEE, October 1993.

[Pr86] V.R. Pratt. "Modeling Concurrency with Partial Orders." *International Journal of Parallel Programming.* 15 (1): 33-71. 1986.

[Ra79] 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.

[ReGo93] M. Reiter and L. Gong. "Preventing Denial and Forgery of Causal Relationships in Distributed Systems." *1993 IEEE Symposium on Research in Security and Privacy.*

[RSA78] R. Rivest, A. Shamir and L. Adlemann. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*, 21: 120-126. February 1978.

[SiKs90] M. Singhal and A.D. Kshemkalyani. *An Efficient Implementation of Vector Clocks.* Computer Science Technical Report TR OSU-CISRC-11/90-TR34, Ohio State University. November 1990.

[Sm93] S.W. Smith. *A Theory of Distributed Time.* Computer Science Technical Report CMU-CS-93-231, Carnegie Mellon University. December 1993.

[Sm94] S.W. Smith. *Secure Distributed Time for Secure Distributed Protocols.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation, to appear in Summer 1994.)

[SJT94] 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.

[SmTy91] 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/February 1993.

[StYe85] R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems." *ACM Transactions on Computer Systems.* 3: 204-226. August 1985.

[TaLo91] Tay, Y.C. and W.T. Loke. *A Theory for Deadlocks.* Computer Science Technical Report CS-TR-344-91, Princeton University. August 1991.

[TyYe93] J.D. Tygar and B.S. Yee. "Dyad: A System for Using Physcially 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.)

[Wein87] S.H. Weingart. "Physical Security for the μABYSS System." *Proceedings of the IEEE Computer Society Conference on Security and Privacy.* 1987.

[Wein91] S.H. Weingart. *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses.* IBM, internal use only. March 1991.

[WWAP91] 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 Reaserch Center. March 1991.

[Yee94] B.S. Yee. *Using Secure Coprocessors.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation, to appear in Spring 1994.)