A Protocol for Secure Transactions

Douglas H. Steves
dhs@cs.utexas.edu

Chris Edmondson-Yurkanan
dragon@cs.utexas.edu

Mohamed Gouda
gouda@cs.utexas.edu

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

Secure transactions form the computational basis for electronic commerce. Many forms of commerce depend upon there being a defined and verifiable relationship between messages in a transaction. We have identified three such relational properties: causality, atomicity and isolation.

Causality is a new property. It allows the receiver of a message to deduce and verify the sequence of messages sent and received by the sender prior to that message.

In this paper, we present a secure transaction protocol that provides relational properties in addition to the normal properties of secure messages.

1 Introduction

A *transaction* is a partial order of messages between two processes. Transactions proceed in three successive phases:

initialization The transaction is accepted by both processes and all parameters are initialized.

exchange One or more data messages are exchanged between the two processes.

termination The transaction is concluded, either successfully or unsuccessfully.

A message is secure if and only if the following properties can be established as required:

authentication The receiver of a message can establish the unique identity of the sender.

privacy The message data can only be observed by the sender and receiver.

integrity The message data can only be altered by the sender without detection.

A transaction is secure if and only if all messages in the transaction are secure and the following relational properties can be established as required:

causality The receiver of a data message can deduce and verify the sequence of previous messages that were sent or received by the sender.

atomicity Either all messages in a transaction may be referenced by other transactions, or none may be.

isolation The messages in a transaction cannot be referenced by other transactions until and unless the first transaction terminates successfully.

A *secure message* protocol is used to exchange singular messages between two processes with one or more of the secure message properties. A *secure transaction* protocol, on the other hand, is used to exchange related secure messages between two processes with one or more of the secure transaction properties.

The secure message properties defined above are well-known, and are already provided in several existing security protocols, such as SET [16], PGP [12] and SSL [3].

The relational properties of atomicity and isolation are also familiar from classical database theory [5], where they provide for computational separation of transactions. In secure transactions we seek a similar effect, so that transactions that involve exchanges of things of value (cash or commodities) occur atomically (all exchanges occur or none do) and in isolation (no item can be exchanged unless it is received in a successful transaction).

The other relational property, causality, allows each process in a transaction to deduce and verify the order in which data messages were sent and received in the transaction. If some of these data messages constitute a negotiation between the processes of the transaction, then causality ensures that each process can prove exactly how the negotiation progressed.

Next we discuss mechanisms that can be used to ensure these relational properties.

1.1 Causality

Ordering is a necessary precondition to ensure causality. Since messages are the only actions in our system, a message can only be caused by a message that preceded it. Causality was first described in [7], using a system of logical clocks.

Ordering can be established by including an authenticated sequence number or virtual time stamp field in each message[14]. This would, however, not be sufficient, since the sequence number in a message would only acknowledge the reception of a certain number of messages prior to the transmission of the message; it would not, however, demonstrate the reception of *specific* messages, since the inclusion of a sequence number or virtual time stamp does not acknowledge the contents of the previous messages.

What is required is a protocol that provides for proof of message reception. Message reception could be proven by including within each message a copy of all previous messages.[8] While this would certainly be sufficient to prove causality, it would also be inefficient, especially for extended transactions. More importantly, it might also violate the privacy policy for multi-party transactions.

Early versions of the Isis system[1] included message digests or hashes as evidence of causality, and this is sufficient to address both of these concerns. In [10], the authors apply the signature technique from [14] to these message digests in order to provide authentication.

Our protocol uses a similar algorithm, including in each message a cryptographically secure message digest of all previous messages sent or received by the sender of the message. This does not violate any privacy concerns, nor is it inefficient since the size of the digest is relatively small and it can be incrementally computed. We define this message digest as *context*.

Unfortunately it is not always possible to provide causality in every transaction. A process P that sends two messages to another process Q and then receives a response from Q cannot validate the context since P cannot know whether or not Q received one or both messages prior to sending its response.

Fortunately it is possible to modify the protocol in order to provide causality in situations like this, by employing a half-duplex protocol that allows both processes to track the state of the transaction consistently. While normally this restriction would be a hindrance to performance, transactions requiring negotiation are essentially half-duplex since each process must wait for the other's latest message in order to formulate its response.

1.2 Atomicity and Isolation

Atomicity and isolation, are ensured in classical database transactions by using a commit protocol. This protocol is used to turn temporary storage into permanent storage - that is, the updates to the transaction's data are not visible until the commit protocol validates the stored data. Note that it is the presence of a commit record in the database log which effectively validates the transaction's data.

With secure transaction protocols, of course, we are not concerned with the correctness of data in persistent storage, but rather that data messages are not used or referred to unless their corresponding transaction has been properly committed. This can be accomplished by:

1) including a transaction identifier within each data message that effectively ties the message to the transaction
2) using a commit protocol to conclude each transaction
3) requiring a valid commit record for a transaction prior to the use of a message from that transaction.

The protocol that we define in this paper addresses the first two of these; the last would require the complete definition of the system to ensure that all message references are properly constrained.

Two other types of secure message atomicity protocols are discussed in [15]. *Encryption-based* protocols guarantee atomicity by having processes encrypt all messages that they send, and then using a trusted third party to distribute the keys once all messages are acknowledged by the receiver. *Authority-based* protocols guarantee atomicity by having processes send all transaction messages to the trusted third party, who will then forward them once all messages in the transaction have been received.

Neither of these is suitable, however, for transactions in which earlier messages must be read before a response can be sent. The use of a database-like commit protocol effectively enables this, since it allows write or send operations to depend upon previous read or receive operations.

In the next section we present a protocol that allows two processes to conduct secure transactions. This protocol extends the privacy, integrity and authentication properties of secure messages to also include causality, atomicity and isolation. This protocol is based on trustful relationships, so the properties are not provable to a trusted third party; they can, however, be confirmed by either party, which allows security violations to be detected prior to commitment of the transaction.

In later work, we extend this protocol to model distrustful and multi-party relationships, where the protocol can be used to prove correct transactions and to disprove incorrect transactions, and we provide an implementation of both models.

2 Secure Transaction Protocol

2.1 Overview

The Secure Transaction Protocol (STP) is a two party protocol that provides authentication, integrity, privacy,[1] atomicity, isolation and causality. In STP, a process may engage in many concurrent transactions with other system processes. Each process in the system has a unique identity (*UID*) and a set of *identity certificates*. The identity certificate used for a transaction is a function of the UID of the other process in the transaction.

Logically, an identity certificate for a process contains the UID and a *signature verification key* for that process. The identity certificate is also used to derive the *signature key* used by the process itself for the transaction. The sender uses the signature key to sign (authenticate) each message, while the receiver uses the verification key to validate that signature. The concepts of signature key and signature verification key map easily to either asymmetric or symmetric cryptography.

Each transaction in the system has a unique identifier (*TID*). For each transaction in which a process participates, the process stores the UID of the other process, the data messages sent and received during that transaction and the state of the transaction. The process also stores the other process's signature verification key to be used during the transaction.

Last, the process maintains the *context* for each active transaction. It is computed as the signed, cryptographic message digest of all data sent or received thus far in the transaction. It is included in all data messages so that the data being sent is effectively represented as the result of previous messages.

During the initialization phase of a normal transaction, one process attempts to begin a transaction with another. The second process may accept or reject the transaction.

If the second process accepts, the transaction enters the exchange phase. The processes begin transmitting data alternately, using a half duplex protocol controlled by a send token.

[1]Privacy has been omitted from the following specification for readability.

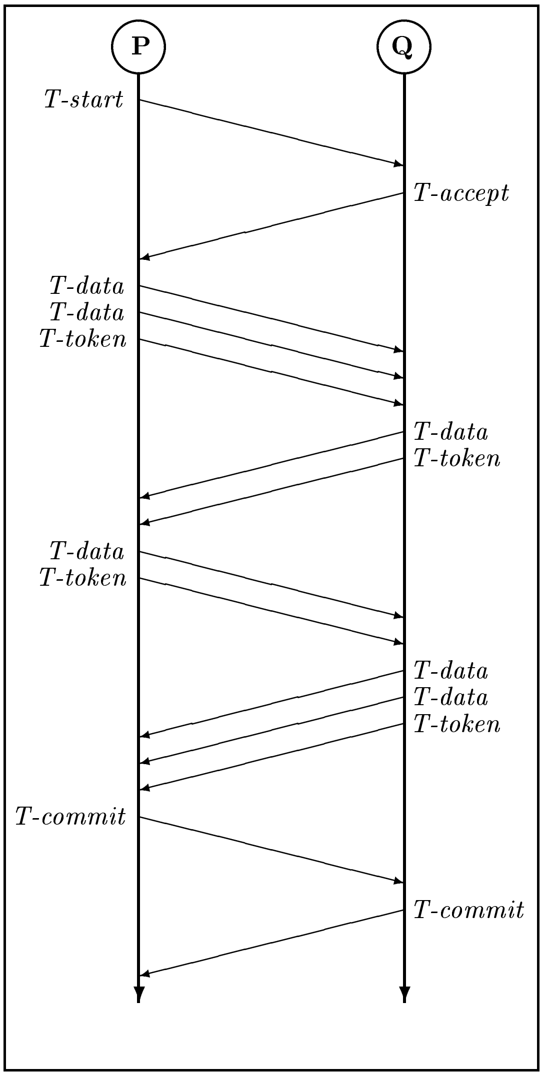

Figure 1: Basic protocol

When the data transfers are complete, the transaction enters the termination phase. If the process with the token wishes to commit the transaction, it sends the other process a commit message. If the second process agrees, it returns its own commit message and the transaction is terminated successfully.

These message transfers are depicted in Figure 1.

This protocol assumes that the underlying message transport mechanism will provide reliable, ordered communication except in the presence of persistent media failures or in the event of an adversarial attack. If the communication channel cannot confirm the correct delivery of a message, it will return a fault exception to the process. Messages deleted by an adversarial process will be detected by the channel, while inserted or altered messages must be detected by the receiving process.

In the following sections, we define **STP** in terms of protocol messages, state and transitions. Messages define the state which can be communicated between processes, state defines discrete and specific points in protocol processing, and transitions define the relationship between messages and state. Lastly, we define the protocol itself, using the Abstract Protocol language, from [4].

2.2 Protocol Messages

Our protocol contains the following message types:

transaction start
The *T-start* message is used to begin a transaction. This message contains:

1) the transaction identifier (TID)
2) the identity certificate of the sender
3) the signature

transaction accept
The *T-accept* message is sent in response to a *T-start* message if the process decides to accept the transaction. This message contains:

1) the TID from the *T-start* message
2) the process's own identity certificate
3) the signature

transaction data
Once the transaction has been established, the processes may exchange data with *T-data* messages. These messages contain:

1) the TID
2) the data
3) the context
4) the signature

transaction token
Communication in **STP** is half duplex. When one process is through sending *T-data* messages, it will send a *T-token* message to allow the other process to transmit. This message contains:

1) the TID
2) the context
3) the signature

transaction commit
When a process wishes to conclude a transaction successfully, it sends a *T-commit* message to the other process. This message contains:

1) the TID
2) the context
3) the signature

transaction abort
When a transaction wishes to terminate a transaction without commitment, it sends a *T-abort* message to the other process. This message contains:

1) the TID
2) the signature

transaction fault
If there is a failure in communications, the communication system will 'send' a *T-fault* message to the process for each message sent by that process which cannot be delivered with the defined semantics. This message contains the TID for the transaction during which the message was sent. Note that the *T-fault* message does not indicate that the message was not transmitted correctly, but only that the communication subsystem cannot verify the correct transmission.

2.3 Protocol States

In **STP**, a process may execute many concurrent transactions. Each transaction is in one of the following states:

null
A transaction is in the null state until the initialization exchange is completed. In our model, this is the starting state for all transactions.

sending, receiving
A transaction which has been successfully initialized and is currently active (its processes are currently exchanging messages) will be either in the *sending* or *receiving* states.

ready
When a process wishes to commit a transaction, the transaction enters the *ready* state. This signifies that data transfer is over. Note that only one process (the one that tries to commit the transaction) will enter this state.

c**committed**
If both processes agree to end the transaction successfully, the transaction will enter the *committed* state, a terminal state.

aborted
The other terminal state for a transaction is the *aborted* state. This signifies that a transaction failed for some reason.

2.4 Protocol State Transitions

Transactions move between states according to messages sent or received. Invalid messages (those without a verifiable signature or that are not acceptable in the current state of the transaction) are discarded without affecting the state of the transaction.

null state
Processes begin each transaction in the *null* state.

For a transaction in this state, processes may send the following message:

T-start will cause the transaction to remain in the *null* state.

Processes may receive the following messages:

T-start may be accepted or rejected. If the transaction is accepted, the transaction moves into the *receiving* state; if it is rejected, the transaction remains in the *null* state.

T-accept will cause the transaction to move into the *sending* state if the message was sent in response to a *T-start* message sent previously by the process.

T-fault has no effect on the process.

sending state

If a transaction is in the *sending* state, a process may send the following messages:

T-data will cause the transaction to remain in the *sending* state.

T-token will cause the transaction to move to the *receiving* state.

T-abort will cause the transaction to move to the *aborted* state.

T-commit will cause the transaction to move to the *ready* state.

Processes may receive the following message:

T-fault will cause the transaction to move to the *aborted* state.

receiving state

For a transaction in the *receiving* state, a process may receive the following messages:

T-data will cause the transaction to remain in the *receiving* state.

T-token will cause the transaction to move to the *sending* state.

T-abort will cause the transaction to move to the *aborted* state.

T-commit enables the process to commit or abort the transaction. Sending:

- *T-commit* will cause the transaction to move to the *committed* state.
- *T-abort* will cause the transaction to move to the *aborted* state.

T-fault will cause the transaction to move to the *aborted* state.

ready state

For a transaction in the *ready* state, a process may send no new messages and may receive the following messages:

T-abort will cause the transaction to move to the *aborted* state.

T-commit will cause the transaction to move to the *committed* state.

T-fault will cause the transaction to remain in the *ready* state and to resend *T-commit*.

committed, aborted states

A transaction in either the *committed* or *aborted* states is in a terminal state.

The state machine for **STP**, excluding the *T-fault* message type, is depicted in Figure 2.

2.5 Formal Specification

Abstract Protocol Language

Both processes execute the same protocol. A process consists of various types of data declarations and a set of guarded commands. A guarded command consists of a predicate, which serves to enable the command, and an action, which will be executed atomically. The set of guarded commands is evaluated repeatedly, and at each execution an enabled command is executed. If multiple commands are enabled, then one is selected non-deterministically, but under the following fairness condition: a command that is enabled continuously will eventually be executed.

Selection statements (**if** ... □ ... □ ... **fi**) also have the above non-deterministic and fairness properties.

One data declaration type should be mentioned: a **parameter** datum is generated each time the guarded command set is evaluated; the parameter is set to a random value of the same type as the parameter.

Annotations

The STP process is passed its user identifier **self** as a process parameter and receives as input an array of identity certificates, **self_cert**. These certificates are indexed according to user identifier. A certificate is used as a parameter to the **SelfKey** function, which computes the key that the process uses to sign messages for the transaction. Messages may be signed with the **Sign** function.

Processes exchange certificates during the transaction initiation phase. A process that receives a certificate will use the **Id** and **Key** functions to extract the user identifier and the verification key from the certificate. The verification key is then used as a parameter to the **Verify** function, which validates the signature computed by the sender.

Transaction state is kept in arrays indexed according to transaction identifier. The **tid** datum is a process parameter that is reset each time the guarded

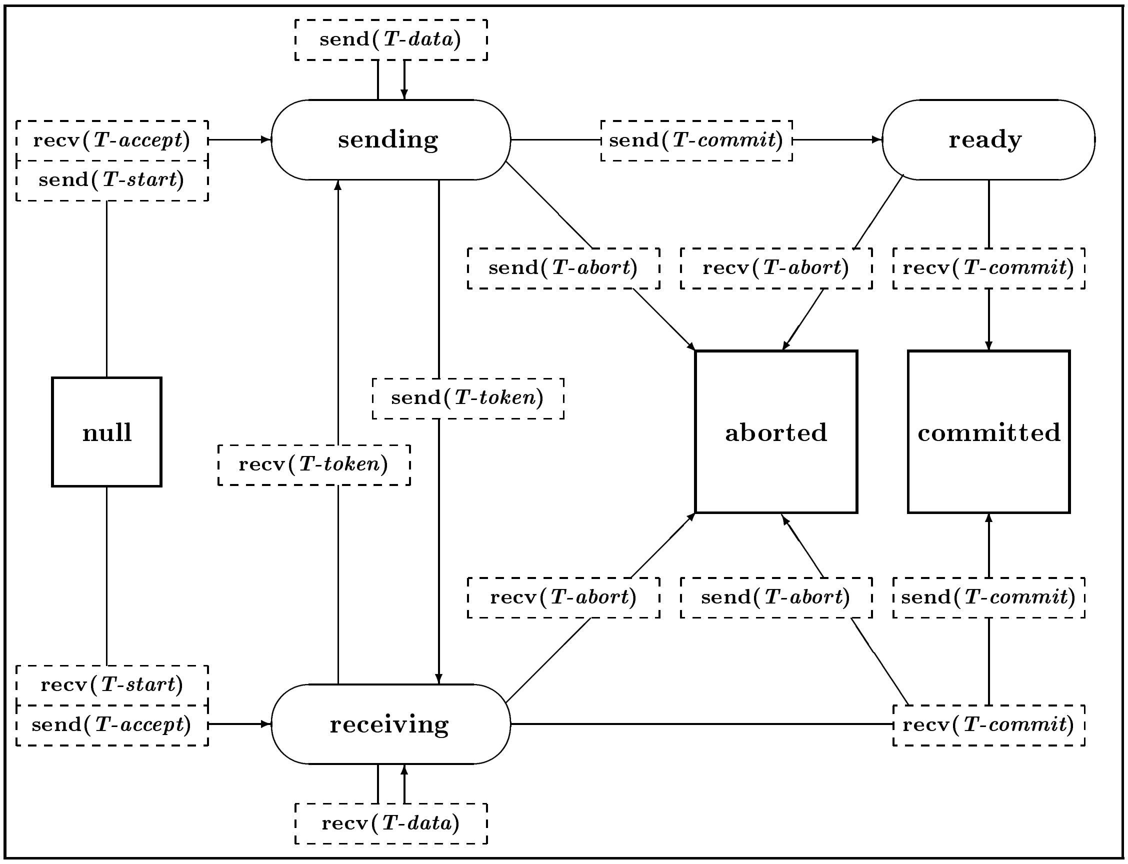

Figure 2: STP State Machine

command set is evaluated, and so will either be a random number or will be equal to the TID field of a received message. If the former, its uniqueness can be determined with the **Unique** function, which implicitly registers TIDs and so prevents their reuse.

The transaction state will be initialized after the transaction has been accepted, and then is updated during the subsequent exchange and termination phases. We assume that data storage is persistent and durable.

For each transaction of which the process is a member, it will maintain the following information:

t_state the current state (initially *null*)
t_other the identity of the other process
t_key the verification key
t_data the data sent or received
t_seq the current message number
t_context the current context

The context is recomputed each time data is sent or received, using the **Hash** function. This function computes a cryptographic message digest of the concatenation of its arguments.

The sections below annotate the code for each action in the process. Note that the temporary boolean variable **valid** is computed for each received message, and is true if the signature is verified and if the transaction state allows this message. Invalid messages are always dropped with no other effect.

Note as well that the signature always includes the message type, so the message body can't be used to forge a different type of message, and that the signature does not cover certificates, since these have independent integrity checks.

Action 0

In this section, a process attempts to begin a new transaction. If **tid** is unique and the local state for this transaction is *null*, the process will non-deterministically decide either to **skip** or to initiate the transaction.

If it decides to initiate the transaction, it selects a process (at random) along with an identity certificate corresponding to the UID of that process. It then sends a *T-start* message to the other process with the identity certificate.

Action 1

A process that receives a valid *T-start* message can decide to accept or reject the transaction. Rejected transactions are simply dropped - the transaction will also remain in the *null* state for the other process.

To accept the transaction, the process selects the identity certificate corresponding to the UID of the initiating process, and sends a *T-accept* message to the other process with that certificate. It then initializes the transaction variables and sets the state for the transaction to *receiving*.

Action 2

When the transaction initiator gets a valid *T-accept* message, it verifies that it was sent by the process to whom it sent the *T-start* message. If so, it also initializes the variables for that transactions, but sets its state to *sending*.

Action 3

A process with a transaction in the *sending* state will non-deterministically either send data, pass the token, commit the transaction or abort the transaction.

In order to send data, a process sends a *T-data* message. The current context is included with each data message to establish causality. After sending the message, the data is stored, the sequence number is updated for the next message send or receive, and the context will be updated with the new data to reflect the current state of the transaction.

When a process decides to pass the token to the other process, it sends a *T-token* message and sets the state of the transaction to *receiving*.

When a process decides to commit the transaction, it sends a *T-commit* and sets the state of the transaction to *ready*.

When a process decides to abort the transaction, it sends a *T-abort* message and sets the state of the transaction to *aborted*.

Action 4

When a process receives a valid *T-data* message, it discards the data if the received context is incorrect; otherwise, it stores the data and updates the sequence number for the next message send or receive.

Action 5

When a process receives a valid *T-token* message, it will abort the transaction if the received context is incorrect; otherwise it sets its state to *sending*.

Action 6

When a process receives a valid *T-commit* message, it will process the message according to the state of the transaction.

If the transaction is already *aborted* or *committed*, the process's original *T-abort* or *T-commit* message was probably lost. and so the process resends the appropriate message.

If the transaction is in the *ready* state, then the message is in response to the process's earlier commit request, and the process sets the transaction state to *committed*.

If the transaction is in the *receiving* state, then the process non-deterministically either aborts or commits the transaction, and sets the transaction state accordingly. It will only commit the transaction if the received context matches its own, but note that it does not have to do so. If it does choose to accept the commit request, it returns a *T-commit* message that has a null data context field. Note that by accepting the commit, it implicitly asserts that its transaction context is equal to the received context, and so there is no reason to return its own value for context.

Action 7

When a process receives a valid *T-abort* message for a transaction, it sets the state of that transaction to *aborted*.

Action 8

When a process receives a *T-fault* message for a transaction, it processes it according to the state of that transaction. If the transaction is active (*sending* or *receiving*), the transaction is aborted. If the transaction is in the *ready* state, it is waiting for a response to its commit request, and so the process resends it. Otherwise the process does nothing.

STP

```
process STP[self : 0..MAXUID]
constant
        MAXUID,
        MAXTID,
        MAXSEQ,
        MAXCERT,
        MAXSIGN,
        MAXHASH,
        MAXKEY
input
        self_cert :  array [0..MAXUID] of 0..MAXCERT
function
        Sign(0..MAXKEY, integer) returns 0..MAXSIGN,
        Verify(0..MAXKEY, integer) returns 0..MAXSIGN,
        Hash(integer) returns 0..MAXHASH,
        Unique(0..MAXTID) returns boolean,
        Id(0..MAXCERT) returns 0..MAXUID,
        Key(0..MAXCERT) returns 0..MAXKEY,
        SelfKey(0..MAXCERT) returns 0..MAXKEY
variable
        msgtype :   set {T-start, T-accept, T-data, T-token, T-commit, T-abort, T-fault},
        state :     set {null, sending, receiving, ready, committed, aborted},
        t_state :   array [0..MAXTID] of state init null,
        t_other :   array [0..MAXTID] of 0..MAXUID,
        t_key :     array [0..MAXTID] of 0..MAXKEY,
        t_data :    array [0..MAXTID, 0..MAXSEQ] of integer,
        t_seq :     array [0..MAXTID] of 0..MAXSEQ,
        t_context : array [0..MAXTID] of 0..MAXHASH,
        cert :      0..MAXCERT,
        data :      integer,
        sign :      0..MAXSIGN,
        context :   0..MAXHASH,
        valid :     boolean
parameter
        tid :       0..MAXTID
begin
        (t_state[tid] = null ∧ Unique(tid)) ⇒ {Action 0}
            t_other[tid] := any;
            cert := self_cert[t_other[tid]];
            sign := Sign(SelfKey(cert), T-start||tid||t_other[tid]);
            send T-start(tid, sign, cert) to t_other[tid];
□       rcv T-start(tid, sign, cert) ⇒ {Action 1}
            valid := (sign = Verify(Key(cert), T-start||tid||self) ∧ t_state[tid] = null);
            if true →
                    skip;
            □ (valid) →
                    t_state[tid], t_seq[tid], t_context[tid] := receiving, 0, 0;
                    t_other[tid], t_key[tid] := Id(cert), Key(cert);
                    cert := self_cert[t_other[tid]];
                    sign := Sign(SelfKey(cert), T-accept||tid);
                    send T-accept(tid, sign, cert) to t_other[tid];
            fi
```

```
□       rcv T-accept(tid, sign, cert) ⇒ {Action 2}
            valid := (sign = Verify(Key(cert), T-accept||tid) ∧ t_state[tid] = null);
            if  (∼valid ∨ ∼(Id(cert) = t_other[tid])) →
                    skip;
            □  (valid ∧ Id(cert) = t_other[tid]) →
                    t_state[tid], t_seq[tid], t_context[tid] := sending, 0, 0;
                    t_key[tid] := Key(cert);
            fi
□       (t_state[tid] = sending) ⇒ {Action 3}
            cert := self_cert[t_other[tid]];
            if  true →
                    data := any;
                    sign := Sign(SelfKey(cert), T-data||tid||data||t_context[tid]);
                    send T-data(tid, data, t_context[tid], sign) to t_other[tid];
                    t_data[tid, t_seq[tid]] := data;
                    t_context[tid] := Hash(data||t_context[tid]);
                    t_seq[tid] := t_seq[tid] + 1;
            □  true →
                    sign := Sign(SelfKey(cert), T-token||tid||t_context[tid]);
                    send T-token(tid, t_context[tid], sign) to t_other[tid];
                    t_state[tid] := receiving;
            □  true →
                    sign = Sign(SelfKey(cert), T-abort||tid);
                    send T-abort(tid, sign) to t_other[tid];
                    t_state[tid] := aborted;
            □  true →
                    sign = Sign(SelfKey(cert), T-commit||tid||t_context[tid]);
                    send T-commit(tid, t_context[tid], sign) to t_other[tid];
                    t_state[tid] := ready;
            fi
□       rcv T-data(tid, data, context, sign) ⇒ {Action 4}
            valid := (sign = Verify(t_key[tid], T-data||tid||data||context) ∧ t_state[tid] = receiving);
            if  (∼valid ∨ ∼(context = t_context[tid])) →
                    skip;
            □  (valid ∧ context = t_context[tid]) →
                    t_data[tid, t_seq[tid]] := data;
                    t_context[tid] := Hash(data||t_context[tid]);
                    t_seq[tid] := t_seq[tid] + 1;
            fi
□       rcv T-token(tid, context, sign) ⇒ {Action 5}
            valid := (sign = Verify(t_key[tid], T-token||tid||context) ∧ t_state[tid] = receiving);
            if  (∼valid) →
                    skip;
            □  (valid ∧ ∼(context = t_context[tid])) →
                    cert := self_cert[t_other[tid]];
                    sign := Sign(SelfKey(cert), T-abort||tid);
                    send T-abort(tid, sign) to t_other[tid];
                    t_state[tid] := aborted;
            □  (valid ∧ context = t_context[tid]) →
                    t_state[tid] := sending;
            fi
□       rcv T-commit(tid, context, sign) ⇒ {Action 6}
            valid := (sign = Verify(t_key[tid], T-commit||tid||context)
                ∧ ∼(t_state[tid] = sending ∨ t_state[tid] = null));
```

```
          if (∼valid) →
               skip;
          □ (valid ∧ t_state[tid] = aborted) →
               cert := self_cert[t_other[tid]];
               sign := Sign(SelfKey(cert), T-abort‖tid);
               send T-abort(tid, sign) to t_other[tid];
          □ (valid ∧ t_state[tid] = committed) →
               cert := self_cert[t_other[tid]];
               sign := Sign(SelfKey(cert), T-commit‖tid‖0);
               send T-commit(tid, 0, sign) to t_other[tid];
          □ (valid ∧ t_state[tid] = ready) →
               t_state[tid] := committed;
          □ (valid ∧ t_state[tid] = receiving) →
               if true →
                    cert := self_cert[t_other[tid]];
                    sign := Sign(SelfKey(cert), T-abort‖tid);
                    send T-abort(tid, sign) to t_other[tid];
                    t_state[tid] := aborted;
               □ (context = t_context[tid]) →
                    cert := self_cert[t_other[tid]];
                    sign := Sign(SelfKey(cert), T-commit‖tid‖0);
                    send T-commit(tid, 0, sign) to t_other[tid];
                    t_state[tid] := committed;
               fi
          fi
□      rcv T-abort(tid, sign) ⇒ {Action 7}
          valid := (sign = Verify(t_key[tid], T-abort‖tid)
             ∧ ∼(t_state[tid] = null ∨ t_state[tid] = committed));
          if (∼valid) →
               skip;
          □ (valid) →
               t_state[tid] := aborted;
          fi
□      rcv T-fault(tid) ⇒ {Action 8}
          if (t_state[tid] = null ∨ t_state[tid] = committed ∨ t_state[tid] = aborted) →
               skip;
          □ (t_state[tid] = sending ∨ t_state[tid] = receiving) →
               t_state[tid] := aborted;
          □ (t_state[tid] = ready) →
               cert := self_cert[t_other[tid]];
               sign := Sign(SelfKey(cert), T-commit‖tid‖t_context[tid]);
               send T-commit(tid, t_context[tid], sign) to t_other[tid];
          fi
end
```

3 Related Work

In this section, we describe several protocols currently in use or under development. All of these provide the properties that are required for secure messages; none of them, however, provide the relational properties that are required for secure transactions.

PGP and PEM

Pretty Good Privacy (PGP) and Privacy Enhanced Mail (PEM) are used mainly for electronic mail security. [12] They are datagram-like in that they allow users to send single, asynchronous messages to each other.

PEM is more flexible, in that it offers a choice of cryptographic algorithms and certificate authorities, although these can't be negotiated since messages are asynchronous. PEM is also more extensible, since PEM data are passed in named header fields, as specified in [2]. PGP requires less infrastructure and is thus suited for more ad hoc environments.

SSL and S-HTTP

Secure Sockets Layer (SSL) [3] and Secure Hypertext Transport Protocol (S-HTTP) [11] allow for the exchange of multiple messages between two processes. The main difference between these protocols and PGP and PEM is that SSL and S-HTTP use a session model, and thus the security mechanisms and parameters used during a session can be negotiated. This allows the degree and kind of security to be varied according to such factors as the nature of the data being exchanged and the vulnerabilities of the underlying communication media. SSL and S-HTTP were designed primarily for WWW-based commerce.

In terms of implementation, SSL fits between the session and transport layers, and is implemented as a replacement for the sockets API to be used by applications requiring secure communications. S-HTTP, on the other hand, is similar to PEM in terms of implementation - its data are passed in named text fields in the HTTP header.

SET

The Secure Electronic Transaction (SET) [16] specification was introduced this year by several credit card and system software vendors. It is currently under design and there are no publicly available reference implementations.

SET uses a session communications model like that of SSL and S-HTTP. SET certificate management is based on X.509, and uses a fixed structure essentially the same as that used for credit card distribution. Both merchants and customers obtain certificates from certificate authorities that represent one or more credit card brands. In order to transact business, a merchant and customer must use certificates obtained from a common brand.

The protocol provides an ad hoc security model that is based on credit card usage. In particular, SET transactions involve only the secure exchange of financial information - negotiations and the actual selection of merchandise are done out of band. Transactions involve a single merchant and consumer engaged in one exchange. The relationship between merchant and customer is asymmetric.

As with the other protocols, SET provides the properties required for secure messages. Note, however, that in recent drafts the specification has defined a linkage function that may be used to insert a reference to one message into another message. The purpose of this function is not completely specified but it could be used to enhance the SET protocol to provide the causality property.

4 Conclusion

In this paper we have presented a protocol for secure transactions that adds causality, atomicity and isolation to the secure message properties of privacy, authentication and integrity that are already provided by current protocols like SET, PGP and SSL. This allows secure transactions to be based on relational properties across messages rather than the properties of single messages.

This protocol is complementary to the current protocols, in that existing elements like identity certificates are used in a compatible manner and new elements like context are orthogonal to the other elements in these protocols. Thus, the new properties can be added easily to these protocols.

References

[1] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[2] D. Crocker. Standard for the format of arpa internet text messages. Technical Report RFC822, IETF, August 1982.

[3] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol v3.0. Internet Draft, March 1996.

[4] Mohamed Gouda. Elements of network protocol design. Contact author for further information.

[5] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1993.

[6] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Netwark Security: Private Communication in a Public World*. Prentice Hall, Englewood Cliffs, NJ, 1995.

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

[8] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. Technical Report TR 88-23, The University of Arizona, Tucson, AZ, 1988.

[9] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[10] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, January 1996.

[11] E. Rescorla and A. Schiffman. The secure hypertext transfer protocol. Internet Draft, December 1994.

[12] Bruce Schneier. *E-Mail Security: How to Keep Your Electronic Messages Private*. John Wiley and Sons, Inc., New York, NY, 1995.

[13] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, Inc., New York, NY, second edition, 1996.

[14] Sean W. Smith and J. D. Tygar. Signed vector timestamps: A secure protocol for partial order time. Technical Report CMU-CS-93-116, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[15] Jiawen Su and J. D. Tygar. Building blocks for atomicity in electronic commerce. In *The 6th Usenix Security Symposium*, pages 97–102. Usenix, July 1996.

[16] VISA, MasterCard, et al. Secure electronic transaction specification, books 1-3. WEB document, June 1996.

[17] Frank Lloyd Wright. *Modern Architecture*. Princeton University Press, Princeton, NJ, 1931.