A FORMAL PRESENTATION OF ELECTRONIC COMMERCE PROTOCOLS

by

Delwin F. Lee

Advisor: Professor Mohamed G. Gouda

Second Reader: Professor Simon S. Lam

COMPUTER SCIENCES HONORS THESIS

CS 379H

The University of Texas at Austin

Spring 2002

A FORMAL PRESENTATION OF ELECTRONIC COMMERCE PROTOCOLS

by Delwin F. Lee
Advised by Professor Mohamed G. Gouda

ABSTRACT

In this paper, we investigate the security of microcommerce digital cash protocols that can facilitate the selling of content over the Internet. Examples of such content are web pages, newspaper articles, and even individual plays of computer games. We focus on two particular digital cash protocols, namely Compaq's Millicent and IBM's Micropayments. For each of these two protocols, we present a formal specification using the Abstract Protocol notation, and then discuss how an adversary can attack the protocol using message forgery, modification, and replay. We then use three concepts of convergence theory, namely closure, convergence, and protection, to show that each protocol is secure against these attacks. Finally, we formally specify and verify the Secure Sockets Layer protocol, which can be used to provide privacy for these digital cash protocols.

TABLE OF CONTENTS

1 Introduction

As commerce on the Internet grows, there is an increasing need for protocols to facilitate online transactions. Many such protocols have been introduced to address this need. For instance, the Secure Electronic Transactions protocol, widely known as SET, enables online transactions by securing purchases made with credit cards [13].

Other protocols, called digital cash protocols, are tailored to small purchases in microcommerce applications. Thus, they are useful in facilitating the selling of content, such as web pages, over the Internet. Examples of digital cash protocols are PayWord and MicroMint [12], Compaq's Millicent [2][9] and IBM's Micropayments [5]. These protocols need to be regarded as "secure" before they can win the approval of customers and vendors alike. In spite of this, none of these protocols have been formally specified and verified. In this paper, we address this issue by formally specifying Compaq's Millicent and IBM's Micropayments, and then use concepts from convergence theory (originally presented in [4]) to verify the correctness and security of these protocols. We chose these two protocols for two reasons. First, they are both prominent and supported by large corporations. Second, the techniques that we use in specifying and verifying these protocols can be used in specifying and verifying other digital cash protocols.

The rest of this paper is organized as follows. In section 2, we present a brief introduction to the Abstract Protocol notation. In section 3, we outline a method used to verify the security of protocols presented in this notation. In section 4, we formally specify the Millicent protocol, while in section 5, we present a proof of its correctness (the formal details of which are in Appendix A). Similarly, in section 6, we formally specify the Micropayments protocol, while in section 7, we present a proof of its correctness (the formal details of which are in Appendix B). In section 8, we specify the Secure Sockets Layer protocol, which can be used to provide privacy in a digital cash protocol; in section 9, we present a proof of this protocol's correctness (the formal details of which are in Appendix C). Finally, in section 10, we present concluding remarks.

2 Abstract Protocol Notation

We specify digital cash protocols using a version of the Abstract Protocol notation presented in [3]. In this notation, each process in a protocol is defined by sets of constants, variables, parameters, and actions. For instance, in a protocol consisting of two processes p and q and two opposite-direction channels, one from p to q and one from q to p, process p can be defined as follows:

```
process p
const  <name of constant>  :      <type of constant>,
       ...
       <name of constant>  :      <type of constant>
inp    <name of input>     :      <type of input>,
       ...
       <name of input>     :      <type of input>
var    <name of variable>  :      <type of variable>,
       ...
       <name of variable>  :      <type of variable>
par    <name of parameter> :      <type of parameter>,
       ...
       <name of parameter> :      <type of parameter>
begin
       <action>
|      <action>
       ...
|      <action>
end
```

The constants of process p have fixed values. Inputs of process p can be read, but not updated, by the actions of process p. Variables of process p, on the other hand, can be read and updated by the actions of process p. Comments can be added anywhere in a process definition; every comment is placed between the two brackets { and }.

Each <action> of process p is of the form:

```
<guard>       ->     <statement>
```

The guard of an action of p is of one of the following three forms: a boolean expression over the constants and variables of p, a receive guard of the form **rcv** <message> **from** q, or a timeout guard that contains a boolean expression over the constants and variables of every process and the contents of all channels in the protocol.

A parameter declared in a process is used to write a finite set of actions as one action, with one action for each possible value of the parameter. For example, if process p has the following variable x and parameter i:

```
var    x      :      integer
par    i      :      0 .. n-1
```

then the following parameterized action in process p:

```
x = i         ->     x := x + i
```

is shorthand notation for the following n actions:

```
        x = 0          ->      x := x + 0
|       ...
|       x = n-1        ->      x := x + n-1
```

Executing an action consists of executing the statement of this action. Executing the actions of different processes in a protocol proceeds according to the following three rules. First, an action is executed only when its guard is true. Second, the actions in a protocol are executed one at a time. Third, an action whose guard is continuously true is eventually executed.

The <statement> of an action of process p is a sequence of <skip>, <send>, <assignment>, <selection>, or <iteration> statements of the following forms:

```
        <skip>          :       skip
        <send>          :       send <message> to q
        <assignment>    :       <variable in p>             :=      <expression>
        <selection>     :       if <boolean expression>     ->      <statement>
                                ...
                                |  <boolean expression>     ->      <statement>
                                fi
        <iteration>     :       do <boolean expression>     ->      <statement>
                                od
```

Executing an action of process p can cause a message to be sent to process q. There are two channels between the two processes: one is from p to q, and the other is from q to p. Each sent message from p to q remains in the channel from p to q until it is eventually received by process q. Messages that reside simultaneously in a channel form a sequence and are received, one at a time, in the same order in which they were sent.

Finally, our protocol specifications include process arrays. A process array is a finite set of processes; each process has the same set of constants, variables, inputs, parameters, and actions. Therefore, all processes in a process array can be defined by a single representative process in the array. For instance, the following declaration:

```
        process c[i: 0..m-1]
```

declares a process array c that consists of the m processes c[0], c[1], ..., c[m-1].

3 Closure, Convergence, and Protection

In this section, we outline a method for verifying the security of protocols that are specified using the Abstract Protocol notation. Later in this paper, we use this method to verify the security of Millicent and Micropayments. This verification method is based on the following definitions.

A *state* of a protocol is a function that assigns to each variable in each process in the protocol a value from its domain of values, and assigns to each channel in the protocol a sequence of messages. For simplicity, we assume that at each protocol state p, the guard of at least one action in some process in the protocol is true at p.

Some states of a protocol are called the *initial states* of that protocol.

A *transition* of a protocol is a pair (p, q) of states of the protocol such that some process in the protocol has an action whose guard is true at state p and execution of this action when the protocol is at state p yields the protocol at state q.

A *computation* of a protocol is an infinite sequence (p.0, p.1, p.2, ...) of protocol states such that each pair (p.i, p.(i+1)) of successive states in the sequence is a protocol transition.

A state of a protocol is called a *safe* state if it occurs in any protocol computation (p.0, p.1, p.2, …) where p.0 is an initial state of the protocol.

A state of a protocol is called an *error state* if the protocol can reach this state by the adversary executing one of its actions starting from a safe state of the protocol.

A state of a protocol that is not safe is called an *unsafe* state if it is an error state of the protocol or if it occurs in any protocol computation (p.0, p.1, p.2) where p.0 is an error state of the protocol.

A protocol is called *secure* if it satisfies the following three conditions:

i. *Closure:*
   In each protocol computation whose first state is safe, every state is safe.

ii. *Convergence:*
   In each protocol computation whose first state is unsafe, there is a safe state.

iii. *Protection:*
   In each protocol transition, whose first state is unsafe, the critical variables of the protocol do not change their values.

According to the above definitions, every protocol satisfies the closure condition. Thus, to prove that a protocol is secure, it is sufficient to show that the protocol satisfies both the convergence and protection conditions. (See sections 5, 7, and 9 below.)

4 Specification of Millicent

In Millicent [2][9], there are two types of parties, customers and vendors. Each customer has a scrip from each vendor and can use this scrip over a period of time to purchase content from that vendor. Thus, each scrip is both customer-specific and vendor-specific, and has the following fields:

- identity of the customer for which the scrip was minted.
- identity of the vendor that minted the scrip.
- value (possibly in dollars) of the scrip.

The value of a scrip decreases when the customer makes a buy request to the vendor and increases in value when the customer makes a scrip request to the vendor. In each request (whether a buy or scrip request) the customer includes the current scrip value and the value the customer expects in a new scrip to be created by the vendor when the vendor handles the request. When the customer sends a buy request to the vendor, the vendor usually returns a scrip with a smaller value, which is the original value of the scrip minus the purchase price of the requested item. When the customer sends a scrip request, the vendor usually returns a scrip whose value is greater than the original value of the scrip. Figure 1 is a simple diagram that shows the flow of messages in Millicent:

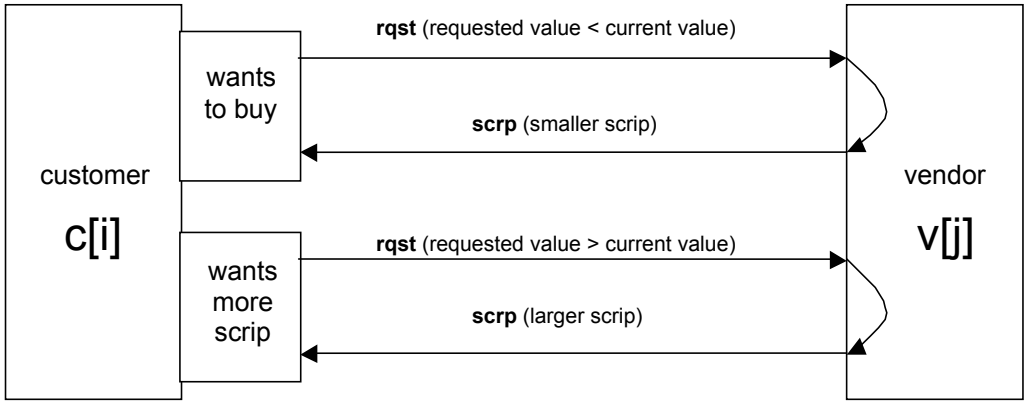


Figure 1. Message flow in Millicent

Because each scrip needs to be protected against adversarial attacks, the scrip contains three additional fields:

- sequence number, used (by the customer and vendor) to detect scrip replay.
- stamp, used (only by the vendor) to detect scrip forgery.
- signature, used (by the customer and vendor) to detect scrip modification.

The customer process c[i] is specified in the Abstract Protocol notation as follows.

```
process c[i: 0..m-1]

inp    sc                  : array [0..n-1] of integer      { shared secrets }
                                                            { sc[j] in c[i] = sc[i] in v[j] }
var    val, seq, stamp     : array [0..n-1] of integer,
       sig                 : integer,
       newval              : integer,
       val', seq', stamp'  : integer,
       sig'                : integer,
       ready               : array [0..n-1] of boolean

par    j                   : 0..n-1

begin
  ready[j]   ->
       ready[j] := false
       newval := any;
       sig    := MD(i | j | val[j] | seq[j] | stamp[j] | newval | sc[j]);
       send rqst(i, j, val[j], seq[j], stamp[j], sig, newval) to v[j];

| rcv scrp(i, j, val', seq', stamp', sig') from v[j]  ->
       if seq[j]+1 ≠ seq'  ->     skip
       |  seq[j]+1 = seq'  ->
              sig := MD(i | j | val' | seq' | stamp' | sc[j]);
              if sig ≠ sig'        ->     skip
              |  sig = sig'        ->
                     val[j]   := val';
                     seq[j]   := seq';
                     stamp[j] := stamp';
                     ready[j] := false;
              fi
       fi

| timeout ~ready[j] ^ #ch.c[i].v[j] = 0 ^ #ch.v[j].c[i] = 0 ->
       ready[j] := true
end
```

Customer c[i] has three actions. In the first, c[i] sends a rqst message to a vendor v[j]. First, c[i] decides upon a new scrip value, newval, to be included in the message. To purchase an item (from v[j]), c[i] chooses newval to be smaller than the scrip's current value. To grow the value of its scrip, c[i] chooses newval to be larger than the scrip's current value. Second, c[i] computes the signature, sig, to be included in the message. The signature is computing by the following statement:

```
       sig    := MD(i | j | val[j] | seq[j] | stamp[j] | newval | sc[j]);
```

Note that the "|" symbol denotes the integer concatenation operator, and MD denotes the message digest function [11]. Thus, the signature is computed by applying the message digest function to the concatenation of the fields of the scrip, newval, and the shared secret, sc[j], between customer c[i] and vendor v[j].

In the second action, c[i] receives a new scrip from v[j]. The value of the scrip usually equals the value that c[i] had requested in its last rqst message to v[j] earlier. Customer c[i] first checks the sequence number of the new scrip. If it is not the one c[i] is expecting, c[i] discards the message. Otherwise, c[i] continues to process the scrip by verifying its digital signature. It does

this by computing its own version of the signature and then comparing it to the one received with the message. If they do not match, c[i] discards the message. Otherwise, c[i] replaces its current scrip with the one in the received message.

In the third action, c[i] detects that ready[j] is false and that the two channels between c[i] and v[j] are empty. This indicates that either the last rqst message from c[i] to v[j] or the last scrp message from v[j] to c[i] was discarded. In this case, c[i] times out and resets ready[j]. This in turn causes c[i] to send a new rqst message.

The vendor process v[j] is specified as follows.

```
process v[j: 0..n-1]

inp    sc                  : array [0..m-1] of integer,     { shared secrets }
       scv                 : integer                        { stamp secret }

var    val, seq, stamp     : array [0..m-1] of integer,
       sig                 : integer,
       newval              : integer,
       val', seq', stamp'  : integer,
       sig'                : integer,
       st                  : integer

par    i                   : 0..m-1

begin
  rcv rqst(i, j, val', seq', stamp', sig', newval) from c[i] ->
       if seq[i]-1 ≠ seq' ^ seq[i] ≠ seq'      ->     skip
       |  seq[i]-1 = seq'  ->
              sig := MD(i | j | val[i] | seq[i] | stamp[i] | sc[i]);
              send scrp(i, j, val[i], seq[i], stamp[i], sig) to c[i]
       |  seq[i] = seq'    ->
              st  := MD(i | j | val' | seq' | scv);
              sig := MD(i | j | val' | seq' | st | newval | sc[i]);
              if st ≠ stamp' v sig ≠ sig'      ->     skip
              |  st = stamp' ^ sig = sig'      ->
                     val[i] := any;
                     seq[i] := seq[i]+1;
                     stamp[i] := MD(i | j | val[i] | seq[i] | scv);
                     sig := MD(i | j | val[i] | seq[i] | stamp[i] | sc[i]);
                     send scrp(i, j, val[i], seq[i], stamp[i], sig) to c[i]
              fi
       fi
end
```

The vendor v[j] has only one action. In this action, v[j] receives a rqst message from a customer c[i], and compares the sequence number in the message with the expected sequence number. Let s be the sequence number in the last scrip that v[j] minted for c[i]. Hence, v[j] expects to see s or s-1 in the received message. There are three cases to consider. In the first, the sequence number in the received message is neither equal to s nor s-1. In this case the received rqst message is an old message being replayed by the adversary, and v[j] discards the message. In the second case, the sequence number in the received message equals n-1. In this case v[j] knows that this request is valid but resent. In other words, the scrp message that v[j] sent earlier

(containing the sequence number s) was not received by c[i], which caused c[i] to time-out and send a new rqst message. In this case, v[j] simply resends the last scrip it minted for c[i].

In the third case, the sequence number in the received message equals s. This means that the request is a new one. In this case, v[j] checks the validity of both the stamp and signature in the received message, and then replies to the message by sending a scrp message to c[i]. Note that although field val[i] in the sent scrp message usually has the same value as field newval in the received rqst message, v[j] has the option of keeping the value of val[i] unchanged to indicate that it has rejected c[i]'s request.

5 Verification of Millicent

In this section, we present a sketch of a correctness proof of the Millicent protocol. Formal details of this proof can be found in Appendix A.

Figure 2 shows the state transition diagram of the Millicent protocol between a customer c[i] and a vendor v[j]. The states in this diagram, namely S.0, S.1, and S.2, are all safe states. The protocol starts in state S.0, the initial state. When c[i] sends a rqst message to v[j] by executing its first action, c.0, the protocol moves to S.1. When v[j] receives the rqst message and sends back the scrp message by executing its only action, v, the protocol moves to S.2. Finally, when c[i] receives the scrp message by executing its second action, c.1, the protocol returns to state S.0.

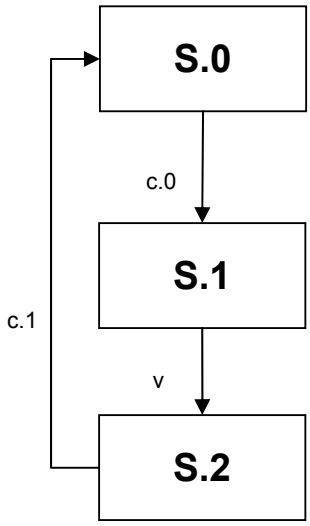


Figure 2. State transition diagram for Millicent

The protocol can be attacked by an adversary capable of executing three attack actions: message forgery, modification, and replay.

Figure 3 shows the state transition diagram of the protocol when the adversary actions are executed. In this diagram, the safe states of the protocol, namely S.0, S.1, and S.2, and the protocol actions, namely c.0, c.1, and v, are as in figure 2. The adversary actions are labeled F (for message forgery), M (for message modification), and R (for message replay). The additional states that result from the adversary actions, labeled U.0 through U.6 are all unsafe states.

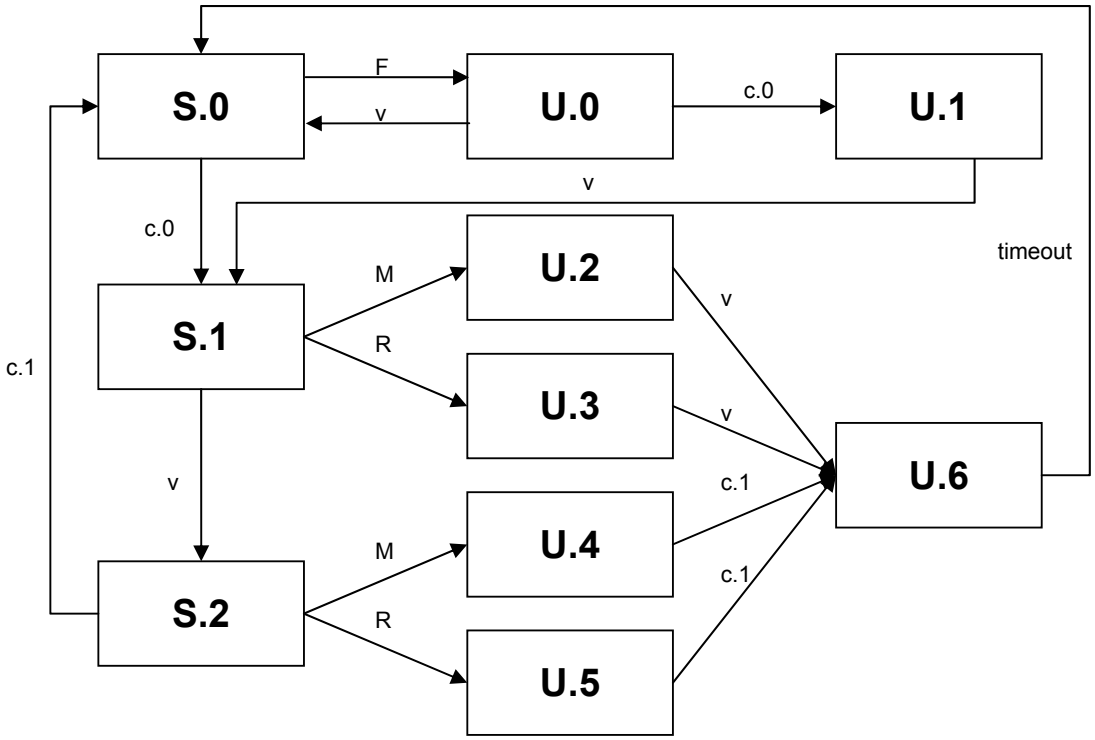


Figure 3. State transition diagram for Millicent (with adversary)

In the message forgery action, the adversary, in collusion with the customer c[i], forges a false scrip and sends it in a rqst message to v[j]. This moves the protocol from safe state S.0 to error state U.0. This attack will fail because when vendor v[j] receives the message, it detects that the message has an invalid stamp (since neither the adversary nor c[i] can forge a valid stamp), and discards it, causing the protocol to return to S.0. Note, however, that if c.0 sends its own rqst message while the protocol is in state U.0, the protocol moves to unsafe state U.1. Then, v[j] receives the forged rqst message and discards it, moving the protocol to state S.1.

In the message modification action, the adversary randomly modifies a message in transit. This can occur in two cases. If c[i]'s rqst message is modified in transit, the protocol moves from safe state S.1 to error state U.2. Likewise, if v[j]'s scrp message is modified in transit, the protocol moves from safe state S.2 to error state U.4. In both cases, the attack will fail since both messages include a digital signature that the receiver can verify. The receiver of a message recalculates its own version of the signature. If the result and the signature sent with the message do not match, the receiver throws the message away, moving the protocol from error state U.2 or U.4 to unsafe state U.6. This causes c[i] to time-out, bringing the protocol back to safe state S.0. Customer c[i] is then free to resend the rqst message.

In the message replay action, the adversary replaces a valid message with a similar message that was sent earlier. As with modification, this can occur in two cases. In the first case,

the current rqst message is replaced with an earlier rqst message, and the protocol moves from safe state S.1 to error state U.3. In the second case, the current scrp message is replaced with an earlier scrp message, and the protocol moves from safe state S.2 to error state U.5. In both cases, the attack will fail. Both c[i] and v[j] remember the sequence number of their the current scrips. When a process receives a message, it checks to see whether the sequence number in the message is what it expects. If this is not the case, the receiving process discards the message, moving the protocol from error state U.3 or U.5 to unsafe state U.6. From U.6, c[i] times-out, bringing the protocol back to safe state S.0.

To show that this Millicent specification is secure against the adversary, we show that it satisfies the two conditions of convergence and protection discussed in section 3.

The convergence condition is satisfied because any computation whose first state is U.0, U.1, U.2, U.3, U.4, U.5, or U.6, has safe state S.0 or S.1 as shown in figure 3.

To show that the protection condition is satisfied, we show that no critical variable is updated when the protocol starts in an unsafe state. Millicent has six critical variables. The first three variables belong to c[i]:

```
var    seq, val, stamp     : array [0..n-1] of integer
```

These three variables are used to store scrip information for each vendor, and are always updated simultaneously. The action that updates the values in these arrays is c.1. In this action, c[i] receives a scrp message from v[j]. Assume that the protocol starts in an unsafe state where the scrp message at the head of the channel between v[j] and c[i] is either modified or replayed. In this case, when c[i] receives the message, it detects that the message is invalid and discards it, without updating any of its three critical variables.

The other three critical variables belong to v[j] and are analogous to the ones described above:

```
var    seq, val, stamp     : array [0..m-1] of integer
```

These three variables can be updated by action v. In this action, v[j] receives and processes a rqst message from c[i]. Again, assume that the protocol starts in an unsafe state where the rqst message in the channel between c[i] and v[j] is either modified or replayed. In this case, when v[j] receives the message, it detects that the message is invalid and discards it, without updating any of its three critical variables. This completes our proof of the security of Millicent against its adversary.

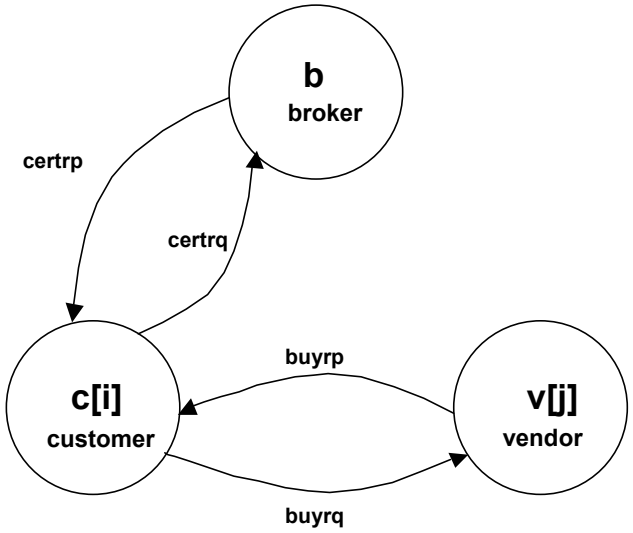
6 Specification of Micropayments

The Micropayments protocol (also known as the MiniPay [5] or NewGenPay protocol) differs greatly from the Millicent protocol. In Millicent, scrip is vendor-specific whereas Micropayments is based on the idea of "universal acceptance by all vendors." Unlike Millicent, Micropayments uses public/private-key operations implemented with RSA [7]. Also, Micropayments does not have a "digital representation" of money. Instead, it uses certificates. There are three types of parties in the Micropayments protocol: customers, vendors, and the broker. Each customer can purchase items from any vendor provided it has a valid certificate from the broker. Each certificate has five fields:

- identity of the customer.
- public key of the customer, used by the vendor to decrypt the customer's order information.
- vendor limit, the amount of money a customer can spend with each vendor in a day.
- timestamp, used by the vendor to check whether the certificate has expired.
- signature, used to ensure integrity of the certificate.

Note that the original specification of the Micropayments protocol calls for the implementation of two distinct spending limits: the vendor limit that specifies the maximum amount of money a customer can spend per vendor per day, and the total limit that specifies the total amount of money a customer can spend per day. In our specification of the protocol, we only implement the vendor limit since the total limit is effectively unenforceable by the vendors or the broker.

When a customer wants to make a purchase from a vendor, it forwards the certificate along with order information to the vendor. (Later, the vendor will deposit these orders with the broker. The broker in turn charges the customer's account.). To make purchases, a customer needs a new certificate every day. This is how the protocol can protect itself against malicious customers; if a customer abuses the system or fails to pay the broker, the broker can refuse to issue new daily certificates to that customer. Figure 4 is a diagram of the flow of messages in Micropayments:



certrq: request for today's certificate
certrp: c[i]'s new certificate, to be used only today
buyrq: purchase request, including order information and certificate
buyrp: requested content (or rejection msg)

Figure 4. Message flow in Micropayments

The broker process b is specified in the Abstract Protocol notation as follows.

```
process b

inp    rk           : integer,                        { private key of broker }
       bkc          : array [0..m-1] of integer,      { pub. keys of customers }
       date         : integer                         { current date }

var    cert         : integer,
       lim          : integer,
       i'           : 0..m-1,
       date', bal'  : integer,
       order'       : integer,

par    i            : 0..m-1,

begin
  rcv certrq(order') from c[i]    ->
       (i', date', bal') := DCR(bkc[i], order');

       if true                    ->     skip
       |  i = i' ^ date = date'   ->
              lim  := any;
              cert := NCR(rk, (i | bkc[i] | date | lim));
              send certrp(cert) to c[i]
       fi
end
```

Broker b has a single action in which it receives a certificate request from any customer c[i]. The broker first decrypts the request information, order', with the customer's public key bkc[i] (which we assume the broker already knows). Then, the broker verifies that the certificate holder's identity matches the identity of the process that sent the certrq message. Second, the

broker checks to see whether the date stamp in the certificate matches the current date. If either of these conditions does not hold, b concludes that the message is invalid and discards it. Otherwise, broker b sends customer c[i] a new certificate to use for the next day. The certificate serves as a symbol of the broker's backing of c[i]. The broker b creates a new certificate for customer c[i] by executing the following statement:

```
cert := NCR(rk, (i | bkc[i] | date | lim))
```

Note that NCR denotes the encryption function. To create the certificate, the broker concatenates the values of the customer's identity, the customer's public key, the current date, and the vendor limit, and then encrypts the result with its private key rk. (Note that the broker still has the option to discard a valid certrq message. For instance, the broker may decide not to approve the customer's request if, for instance, the customer abuses the system or does not pay on time.)

The customer process c[i] is specified as follows.

```
process c[i: 0..m-1]

inp    rk, bk              : integer,          { private and pub keys of c[i] }
       bkb                 : integer,          { public key of broker }
       bkv                 : array [0..n-1] of integer,     { pub keys of ven }
       date                : integer           { current date }

var    cert, cdate         : integer,
       bal, cost           : integer,
       order               : integer,
       i'                  : 0..m-1,
       bk', date', lim     : integer,
       j'                  : 0..n-1,
       cost'               : integer,
       cert', order'       : integer,
       ready               : array [0..n] of boolean
                                               { ready[n] used for broker }
par    j                   : 0..n-1

begin
  ready[n] ^ (∀k: 0 <= k < n: ready[k]) ^ date > cdate      ->
       ready[n] := false;
       order    := NCR(rk, (i | date | bal));
       send certrq(order) to b

| rcv certrp(cert') from b ->
       (i', bk', date', lim) := DCR(bkb, cert');
       if i ≠ i' v bk ≠ bk' v date ≠ date'     ->     skip
       |  i = i' ^ bk = bk' ^ date = date'     ->
              cert     := cert';
              cdate    := date';
              bal      := 0;
              ready[n] := true
       fi

| ready[j] ^ ready[n] ^ date <= cdate   ->
       ready[j] := false;
       cost     := any;
       order    := NCR(rk, (j | date | cost));
       send buyrq(cert, order) to v[j]
```

```
| rcv buyrp(order') from v[j]     ->
       (i', j', date', cost') := DCR(bkv[j], order');

       if i ≠ i' v j ≠ j' v date ≠ date'       ->     skip
       |  i = i' ^ j = j' ^ date = date'       ->     bal      := bal + cost;
                                                     ready[j] := true
       fi

|  timeout ~ready[n] ^ #ch.c[i].b = 0    ^ #ch.b.c[i] = 0          ->
       ready[n] := true

|  timeout ~ready[j] ^ #ch.c[i].v[j] = 0 ^ #ch.v[j].c[i] = 0       ->
       ready[j] := true
end
```

In the first action, customer c[i] sends the broker b, a certificate request. Recall that c[i] needs a new certificate every day it makes a purchase. Hence, this action is executed only when c[i]'s current certificate is at least a day old. To create the request, c[i] encrypts the concatenation of its identity, the current date, and the balance of the previous day's purchases, with its private key:

In the second action, c[i] receives a reply to its certificate request. This certrp message contains a new certificate to be used for purchases today. (We assume that c[i] already knows the public key of the broker.) c[i] first decrypts the certificate with the broker's public key and then checks that the values of i and bk included with the certificate are correct. It also makes sure that the date in the certificate matches the current date. If any of these conditions is false, c[i] discards the message.

In the third action, c[i] sends a buy request to a vendor v[j]. This action is only eligible for execution when c[i]'s current certificate is less than a day old. In this request, c[i] forwards the certificate that it received from b. The customer c[i] also sends order information with this request, which it creates by encrypting the concatenation of the vendor's identity, the current date, and the cost of the requested item, with its private key.

In the fourth action, c[i] receives a buy reply from vendor v[j]; c[i] decrypts the information and then verifies its integrity. If the reply is valid and not replayed, c[i] updates its running balance with the value cost' that vendor v[j] included in the message.

In the fifth action, customer c[i] detects that either its certrq message to the broker b or the corresponding certrp message from b was discarded, so it resets the value of ready[n] to true. This in turn causes c[i] to resend a new certrq message.

Similarly, in the sixth action, c[i] detects that either its buyrq message to the vendor v[j] or the corresponding buyrp message from v[j] was discarded, so it resets the value of ready[j] to true. This in turn causes c[i] to resend a new buyrq message.

The vendor process v[j] is specified as follows.

```
process v[j: 0..n-1]

inp    rk                  : integer,          { this vendor's private key }
       bkb                 : integer,          { public key of broker }
       date                : integer           { current date }

var    bal                 : array [0..m-1] of integer,
       cost                : integer,
       order               : integer,
       i'                  : 0..m-1,
       bkc', cdate', lim   : integer,
       j'                  : 0..n-1,
       date', cost'        : integer,
       cert', order'       : integer,

par    i                   : 0..m-1

begin
  rcv buyrq(cert', order') from c[i]    ->
      (i', bkc', cdate', lim) := DCR(bkb, cert');
      (j', date', cost')      := DCR(bkc', order');

      if i ≠ i' v date ≠ cdate' v j ≠ j' v date ≠ date'     ->     skip
      |  i = i' ^ date = cdate' ^ j = j' ^ date = date'     ->
            if true                    ->     cost := 0
            |  bal[i]+cost <= lim      ->     cost := any
            fi
            bal[i] := bal[i] + cost;
            order  := NCR(rk, (i | j | date | cost));
            send buyrp(order) to c[i]
      fi
end
```

The vendor v[j] has only one action. In this action, v[j] receives a buy request from a customer c[i] and checks the integrity of the received certificate by decrypting it with the broker's public key (which we assume v[j] already knows) and verifying that the values of i and date included in the certificate are valid. Next, v[j] decrypts the customer's order information using c[i]'s public key, included in the certificate, and verifies the order information in a similar manner to the way that it checked the integrity of the certificate. When v[j] concludes that both the certificate and order information are valid, it creates a buyrp message by encrypting the concatenation of the customer's identity, its own identity, the current date, and the cost of the purchased item, with its private key rk. (Note, however, that the value of cost can be 0, indicating that v[j] has "rejected" the buy request. The two most probable reasons are that c[i] exceeds its vendor limit with this purchase, or that the vendor no longer sells the requested item.)

7 Verification of Micropayments

In this section, we present a sketch of a correctness proof of the Micropayments protocol. Formal details of this proof can be found in Appendix B.

Each customer c[i] can be involved in two types of transactions. In the first type of transactions, c[i] interacts with the broker b; in the second type of transactions, c[i] interacts with a vendor. Hence, we present two separate state transition diagrams that express both relationships.

Figure 5 shows the state transition diagram of Micropayments between a customer c[i] and the broker b. The states in this diagram, namely S.0, S.1, and S.2, are all safe states. The protocol starts in S.0. When c[i] sends a certrq message to b by executing its first action c.0, the protocol moves to state S.1. When b receives the certrq message and sends back a certrp message by executing its only action b, the protocol moves to state S.2. Finally, when c[i] receives the certrp message by executing its second action c.1, the protocol moves back to state S.0.

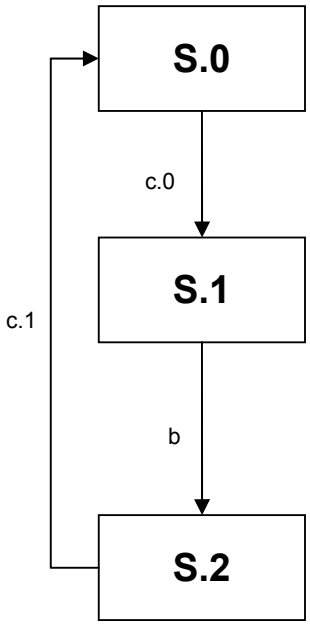


Figure 5. State transition diagram for Micropayments – customer/broker

Figure 6 shows the state transition diagram of Micropayments between a customer c[i] and a vendor v[j]. As before, the protocol begins in state S.0. When c[i] sends a buyrq message to v[j] by executing its third action c.2, the protocol moves to state S.3. When v[j] receives the buyrq and sends back the buyrp message by executing its only action v, the protocol moves to S.4. Finally, when c[i] receives the buyrp by executing its fourth action c.3, the protocol returns to state S.0.

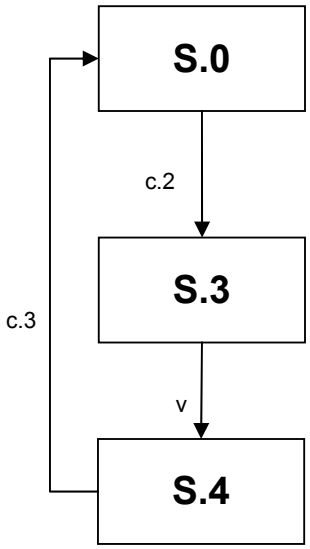


Figure 6. State transition diagram for Micropayments – customer/vendor

As before, we consider an adversary capable of executing three attacks: message forgery, modification, and replay. Figure 7 shows the state transition diagram of the protocol between a customer c[i] and the broker b when these actions are executed. In this diagram, safe states S.0, S.1, and S.2, and protocol actions c.0, c.1, and b, are as in figure 5. The adversary actions are labeled M (for message modification) and R (for message replay). The additional states that result from these actions, namely U.2 through U.6, are all unsafe states.

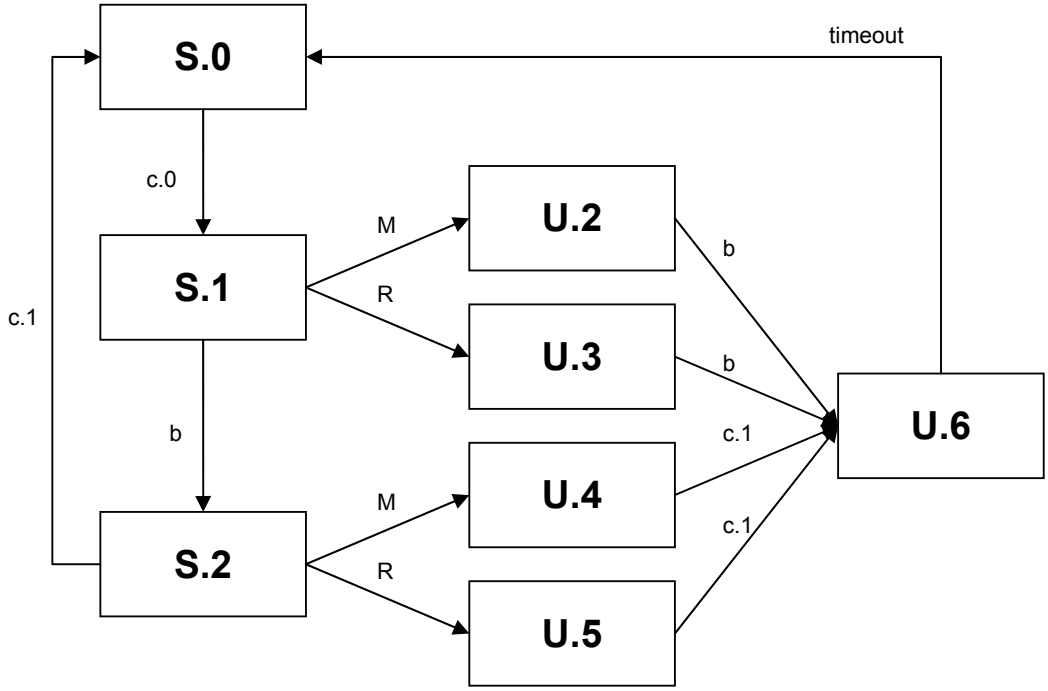


Figure 7. State transition diagram for Micropayments (customer/broker, with adversary)

Similarly, figure 8 shows the state transition diagram of Micropayments between a customer c[i] and a vendor v[j] when the adversary executes its attacks. In this diagram, safe states S.0, S.3, and S.4, and protocol actions c.2, c.3, and v, are as in figure 6. The adversary actions are labeled F (for message forgery), M (for message modification), and R (for message replay). The additional states that result from these actions, namely U.0, U.1, U.7, U.8, U.9, U.10, and U.11 are all unsafe states.

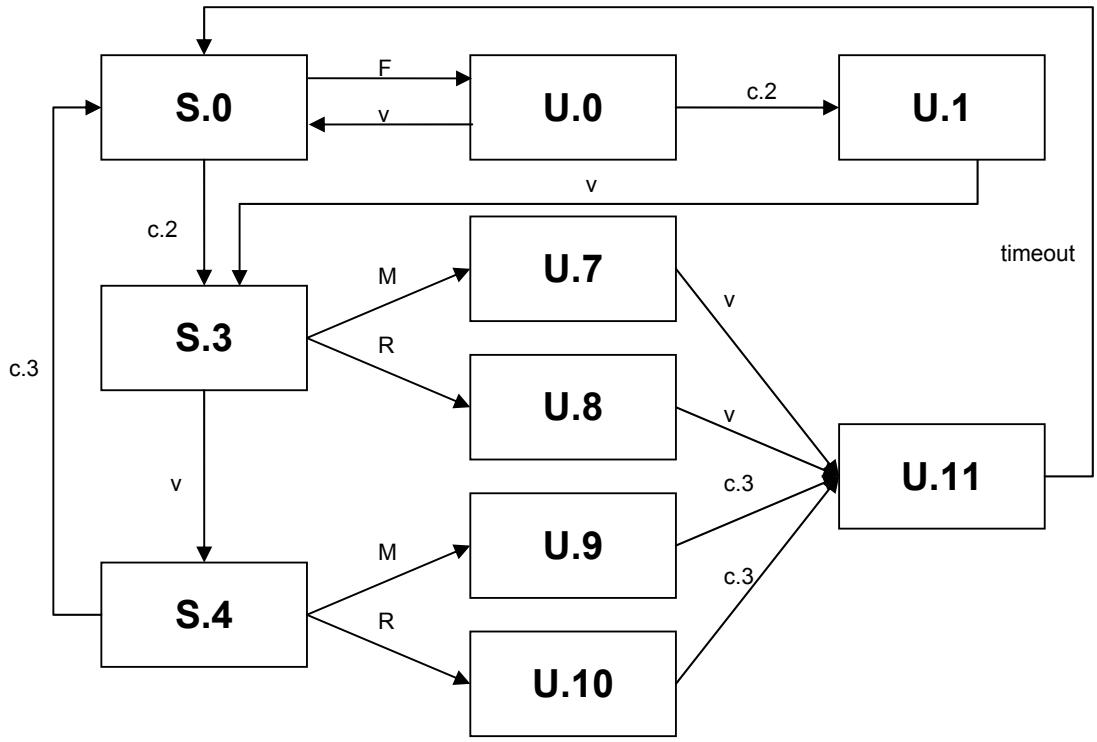


Figure 8. State transition diagram for Micropayments (customer/vendor, with adversary)

In the message forgery action, the adversary, in collusion with c[i], attempts to create its own certificate and sends it in a buyrq message. This moves the protocol from safe state S.0 to error state U.0. This attack will fail because when vendor v[j] receives the message, it detects that the message has an invalid certificate (since neither the adversary nor c[i] have access to the broker's private key, which is necessary to create a valid certificate), and discards it, causing the protocol to return to S.0. Note, however, that if c.0 sends its own buyrq message while the protocol is in state U.0, the protocol moves to unsafe state U.1. Then, v[j] receives the forged rqst message and discards it, moving the protocol to state S.3.

In the message modification action, the adversary randomly modifies a message in transit. This can occur in four cases. If c[i]'s certrq message or b's certrp message is modified while in transit to the other process, the protocol moves from safe state S.1 or S.2 to error state

U.2 or U.4, respectively. Likewise, if c[i]'s buyrq message or v[j]'s buyrp message is modified in transit, the protocol moves from safe state S.3 or S.4 to error state U.7 or U.9, respectively. In all cases, the attack will fail since public/private key encryption is used to ensure message integrity. When a process receives a message, it decrypts it and checks to see if some fields of the message match their expected values. If this is not the case, the receiver throws the message away, moving the protocol from an error state to unsafe state U.6 or U.11. This causes c[i] to time-out, bringing the protocol back to safe state S.0. Afterwards, c[i] is free to resend its request.

In the message replay action, the adversary replaces a valid message at the head of a channel with a similar message sent earlier (before the current day). As with modification, this can occur in four cases. If c[i]'s certrq message or b's certrp message is replaced with an old message while in transit to the other process, the protocol moves from safe state S.1 or S.2 to error state U.3 or U.5, respectively. Likewise, if c[i]'s buyrq message or v[j]'s buyrp message is replaced, the protocol moves from safe state S.3 or S.4 to error state U.8 or U.10, respectively. In all cases, the attack will fail. Every request and reply in the protocol includes a date stamp. When a process receives a message, it inspects the message's date and compares it to the current date. If the request was not originally sent today, the receiver discards the message, again moving the protocol from an error state to unsafe state U.6 or U.11. Finally, c[i] times out, bringing the protocol back to safe state S.0.

To demonstrate that this Micropayments specification is secure against the adversary, we show that it satisfies the two conditions of convergence and protection discussed in section 3.

The convergence condition is satisfied since any computation whose first state is U.0, U.1, U.2, U.3, U.4, U.5, U.6, U.7, U.8, U.9, U.10, or U.11 includes safe state S.0 or S.3. This is evident in figures 7 and 8.

To show that the protection condition is satisfied, we examine our specification's three critical variables. The first two belong to c[i]:

- `cert, ctime : integer,`

cert stores c[i]'s current certificate, and ctime is the timestamp of the certificate. Only action c.1, in which c[i] receives a brand new certificate, can update these two variables (and it does so simultaneously). Assume that the protocol starts in an unsafe state and the certrp message is modified or replayed. When c[i] receives the message, it detects that it is invalid and discards it, without updating either of its critical variables.

The last critical variable belongs to v[j]:

- `bal : array [0..m-1] of integer,`

bal[i] can only be updated by action v.0, in which v[j] receives and processes c[i]'s buy request (buyrq). Assume that the protocol starts in an unsafe state and the buyrq message is modified or replayed. When v[j] receives the message, it detects that it is invalid and throws it away, without updating bal[i].

8 Specification of SSL

The Secure Sockets Layer protocol [1], also known as SSL, can be used by digital cash protocols to provide privacy. Since a system is only as secure as its weakest link, we now provide a formal specification and verification of SSL.

SSL provides three services: client authentication, server authentication, and encrypted connections. In fact, at its core, the protocol is a method of exchanging a shared key between authenticated parties. This key in turn is used to encrypt all messages between the parties. Since we are focused on the use of SSL in the context of digital cash protocols, we make a number of simplifying assumptions in our specification.

1. Our specification implements server authentication but not client authentication. This is done for two reasons. First, in most online commerce applications, we are only concerned with authenticating one side of the transaction (usually the server side). Secondly, client authentication is almost an exact mirror of client authentication, so including both in our specification would yield a great deal of redundancy.
2. The SSL protocol consists of two sub-protocols: the handshake protocol and the record layer protocol. The record layer is responsible for handling the secure transmission of messages after the handshake protocol establishes the secure connection. The specification presented here implements only the handshake protocol.
3. Next, our specification does not include the feature of resumable sessions, since this feature is more of a convenience than a necessity. Thus, our specification does away with the session identifiers.
4. Since certificate distribution is handled by a protocol outside of SSL, we consider it to be an implementation detail. Hence, for simplicity, we assume that all certificates are backed up by a single certificate authority. (SSL certificates are based on the X.509 standard [6].)
5. We also standardize a number of variables in the protocol so that our specification does not need to transmit them in the hello messages. For instance, we assume that the cipher suite will always be RSA. The nonces have also been removed from the hello messages.
6. Furthermore, where possible, we removed some redundancy from several of the cryptographic operations; for instance, some operations that may have used two cryptographic hashes now use one.

Figure 9 shows message flow in our SSL specification.

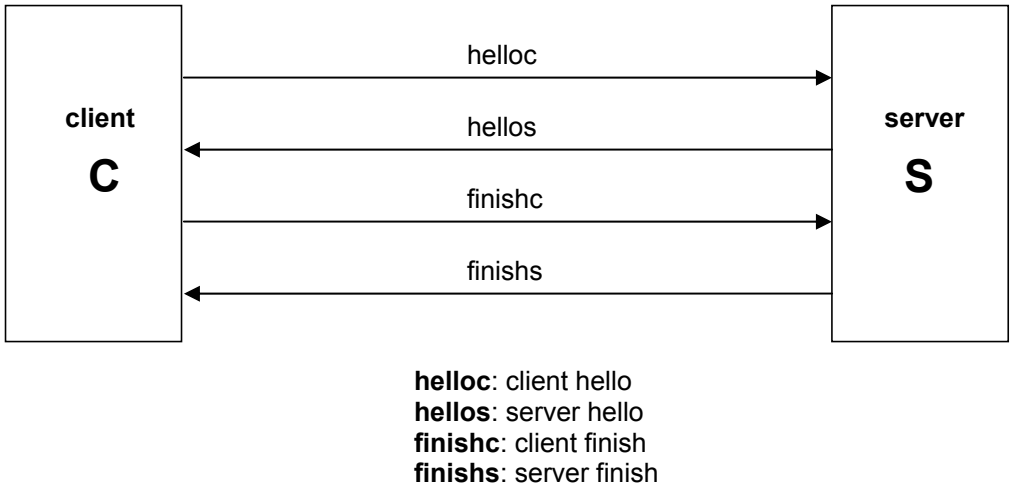


Figure 9. Flow of messages between client and server in SSL

The client process c[i] is specified in the Abstract Protocol notation [3] as follows.

```
process c[i: 0..m-1]

inp    bkca                : integer,          { public key of cert authority }
       date                : integer           { current date }

var    st                  : array [0..n-1] of 0..3,        { current state }
       certs               : array [0..n-1] of integer,
       secret, key         : array [0..n-1] of integer,
       esecret, emsg       : array [0..n-1] of integer,

       certs'              : integer,
       j'                  : 0..n-1,
       bks', date'         : integer,

       msg                 : integer,
       msgs, msgs', emsgs' : integer

par    j                   : 0..n-1

begin
  st[j] = 0   ->     st[j] := 1
                     send helloc to s[j]

| rcv hellos(certs') from s[j]          ->
       (j', bks', date') := DCR(bkca, certs');

       if st[j] ≠ 1 v j ≠ j' v date >  date'    ->     skip
       |  st[j] = 1 ^ j = j' ^ date <= date'    ->
             st[j]         := 2;
             certs[j]      := certs';

             secret[j]     := random;
             esecret[j]    := NCR(bks', secret[j]);

             key[j]        := F(secret[j]);
             msg           := G(i, j, certs[j], secret[j], esecret[j]);
             emsg[j]       := NCR(key[j], msg);
```

```
            send finishc(esecret[j], emsg[j]) to s[j]
        fi

| rcv finishs(emsgs') from s[j]   ->
        msgs'          := DCR(key[j], emsgs');
        msgs           := H(i, j, certs[j], secret[j], esecret[j], emsg[j]);

        if st[j] ≠ 2 v msgs ≠ msgs'              ->     skip
        |  st[j] = 2 ^ msgs = msgs'              ->     st[j] := 3
        fi

| st[j] = 3   ->     st[j] := 0

| timeout st[j] = 1 ^ #ch.c[i].s[j] = 0 ^ #ch.s[j].c[i] = 0        ->
        st[j] := 0

| timeout st[j] = 2 ^ #ch.c[i].s[j] = 0 ^ #ch.s[j].c[i] = 0        ->
        send finishc(esecret[j], emsg[j]) to s[j]
end
```

In the first action, client c[i] checks its state variable st[j] to see whether it is ready to begin a new connection with server s[j]. If this is the case, c[i] updates the state variable and sends a helloc message to the server.

In the second action, c[i] receives the server's hellos message. This begins the server authentication process. The hellos message includes the server's certificate. First, c[i] decrypts the certificate with the public key of the certificate authority (which we assume is already known to all processes); this yields the certificate owner's identity (j'), the public key of the certificate owner (bks'), and the expiration date of the certificate (date').

Next, the client must verify that three conditions hold before proceeding. First, the state variable st[j] must indicate that c[i] was waiting for a hellos message. Second, j' must match the identity of the process that sent the server hello message (j). Third, the current date must be less than or equal to the expiration date of the certificate. If any of these conditions does not hold, c[i] assumes the hellos message is invalid and discards it. Otherwise, server s[j] is authenticated and the client proceeds to generate its finishc message. The client starts by updating the state variable st[j] and then stores away the server's certificate so that it can use it in later stages of the protocol.

Client c[i] then generates a secret and stores it in secret[j]. (This is analogous to the "premaster secret" in the original SSL specifications [1].) It also encrypts this secret with s[j]'s public key bks' and stores the result in esecret[j]. (Note that this step is critical to server authentication; it means that only the real server s[j] can access the secret.) Afterwards, c[i] creates the shared key, a function (F) of secret[j], that will be used during the secure session. Next, c[i] proceeds to generate a signature (msg), which is a function (G) of all the messages sent up to this point in the protocol's execution. This signature is then encrypted with the shared key that was just generated. Finally, c[i] sends the finishc message, consisting of the encrypted secret

(esecret[j]) and the encrypted signature (emsg[j]). Note that the shared key is never transmitted directly to the server. Only the encrypted secret (from which the key can be generated) is sent. The client's finish message also serves as a "secure acknowledgement" to the server.

In the third action, c[i] receives the server's finishs message. (Note that this message is not completely necessary in our specification since we do not implement client authentication. However, to stay true to the original spirit of the SSL protocol, we include it.) Client c[i] starts by decrypting the server's encrypted finish message (emsgs') with the shared key. The client also generates its own version of the finish message by applying function H to the data values it stored earlier. Finally, c[i] verifies that it is in the state where it is waiting for this finish message and verifies that the two versions of the signature match. If either condition does not hold, c[i] discards the message. Otherwise, it updates its state variable (st[j]) to indicate that the handshake is complete.

In the fourth action, c[i] terminates its connection to server s[j] by resetting state variable st[j].

In the fifth action, client c[i] detects that either its helloc message to server s[j] or the corresponding hellos message from s[j] was discarded. As a result, c[i] resets its state variable, causing it to resend a new helloc message.

Similarly, in the sixth action, c[i] detects that either its finishc message to server s[j] or the corresponding finishs message from s[j] was discarded. As a result, c[i] resends the finishc message that it has already generated.

The server process s[j] is specified as follows.

```
process s[j: 0..n-1]

inp    rk, bk              : integer,          { private and pub key of s[j] }
       bkca                : integer,          { pub key of cert authority }
       certs               : integer           { this server's certificate }

var    st                  : array [0..m-1] of 0..1,        { current state }
       key                 : array [0..m-1] of integer,
       emsgs               : array [0..m-1] of integer,

       esecret', emsg'     : integer,

       secret'             : integer,
       key', msg'          : integer,
       msgs                : integer

par    i                   : 0..m-1

begin
  rcv helloc from c[i]     ->
       st[i] := 1;
       send hellos(certs) to c[i]

| rcv finishc(esecret', emsg') from c[i]       ->
```

```
        secret'       := DCR(rk, esecret');
        key'          := F(secret');

        { verify finish message }
        msg           := G(i, j, certs, secret', esecret');
        msg'          := DCR(key', emsg');

        if st[i] ≠ 1 v msg ≠ msg'  ->     skip
        |  st[i] = 1 ^ msg = msg'  ->
               st[i]         := 0;
               key[i]        := key';

               msgs          := H(i, j, certs, secret', esecret', emsg');
               emsgs[i]      := NCR(key[i], msgs);

               send finishs(emsgs[i]) to c[i]
        fi
end
```

In the first action, server s[j] receives a helloc message from client c[i]. The server updates its state variable and sends its certificate in a hellos message to c[i]. (Note that the protocol does not check that the value of its st[i] is 0 before proceeding. However, this is not a security vulnerability, as explained in section 9.)

In the second action, the server receives a finishc message from c[i], which includes the encrypted secret and encrypted signature. The server starts by decrypting the secret with its private key. It then applies function F to the secret to generate the shared key (key'). Using the newly generated key, s[j] then decrypts the encrypted signature. At the same time, the server also generates its own version of the signature using function G. Server s[j] then compares the two versions of the signature to check if they match. It also verifies whether it is in the state in which it expects to receive the helloc message. If either of these conditions does not hold, s[j] discards the message. Otherwise, s[j] updates its state variable, stores the key, and proceeds to generate its own finishs message, consisting of a encrypted signature. This signature is a function (H) of all messages sent up to this point in the protocol.

9 Verification of SSL

In this section, we present a sketch of a correctness proof of the SSL protocol. Formal details of this proof can be found in Appendix C.

Figure 10 shows the state transition diagram between a client c[i] and server s[j]. The states in this diagram, namely S.0 through S.5, are all safe states. The protocol starts in the initial state, S.0. When c[i] sends a helloc message to s[j] by executing its first action c.0, the protocol moves to S.1. When s[j] receives the helloc message and sends back a hellos message by executing its first action s.0, the protocol moves to S.2. When client c[i] receives the hellos message and sends back a finishc message by executing its second action c.1, the protocol moves to S.3. When s[j] receives the finishc message and sends back a finishs message by executing its second action s.1, the protocol moves to S.4. Once c[i] receives the finishs message by executing its third action c.2, the protocol moves to S.5. Finally, when c[i] decides to terminate the secure session by executing its fourth action, c.3, the protocol returns to state S.0.

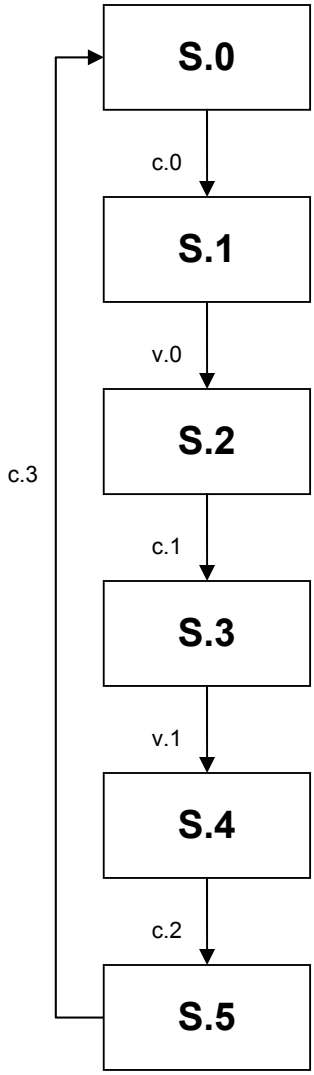


Figure 10. State transition diagram for SSL

The protocol can be attacked by an adversary capable of executing two attack actions: message modification and message replay. (The certificate forgery attack is not considered since it would yield no potential benefit to the adversary.) Figure 11 shows the state transition diagram of the protocol when the adversary actions are executed. Here, the safe states of the protocol, namely S.0 through S.5, and the protocol actions, namely c.0, c.1, c.2, c.3, s.0 and s.1, are as in figure 10. The adversary actions are labeled M (for message modification) and R (for message replay). The states that result from the adversary actions, labeled U.0 through U.6, are all unsafe states.

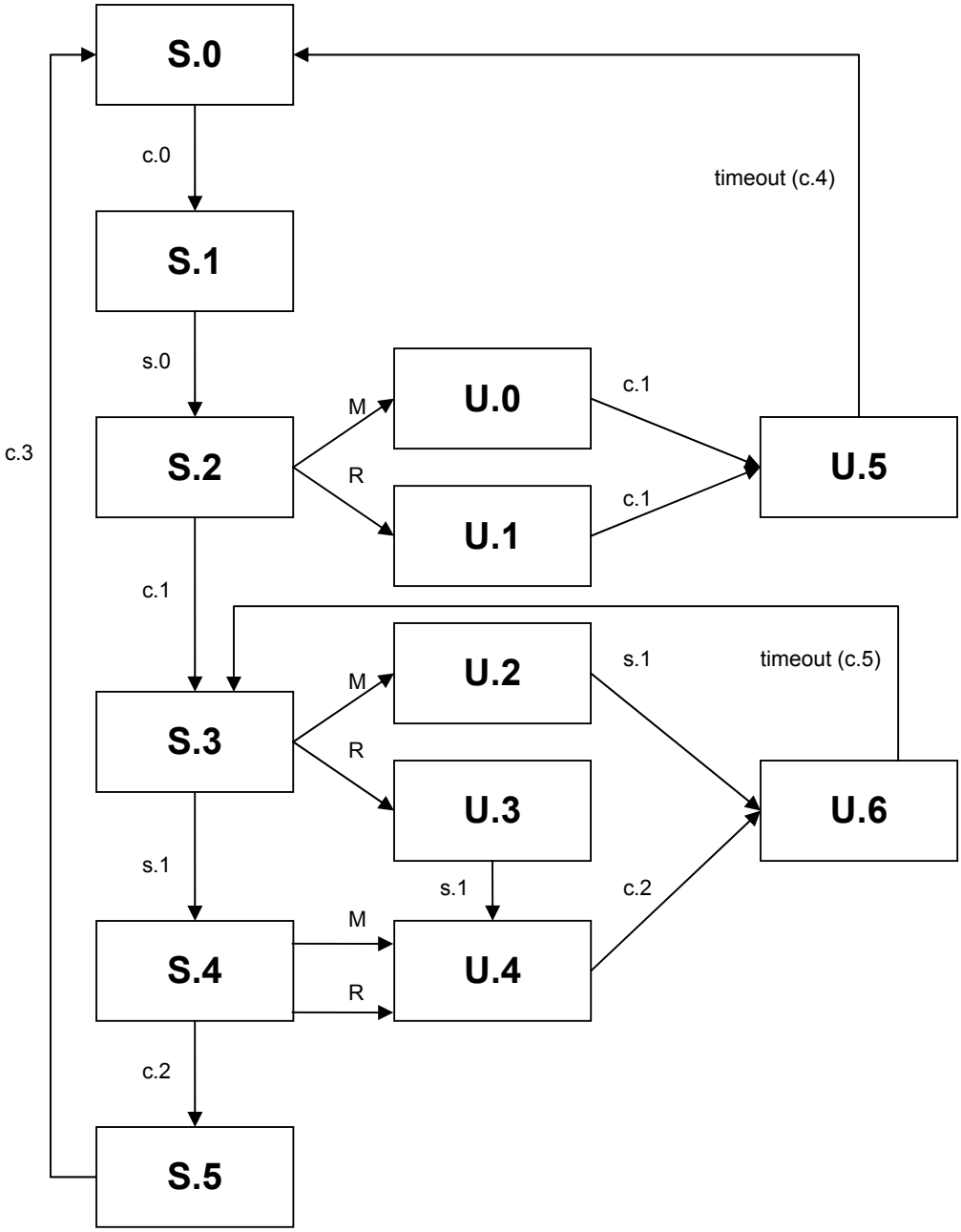


Figure 11. State transition diagram for SSL (with adversary)

Note that no adversary attacks can be executed when the protocol is at safe state S.5. Recall that at S.5, the secure session has been established. There are no messages in the channel to modify or replace. Even if we assume that the adversary could insert a helloc message at this state, in the hopes of terminating the secure session between c[i] and s[j], the protocol would still be secure. This is because the adversary has no way to convince s[j] that it is indeed c[i]; its invalid helloc message would be received at a TCP port different from the one s[j] is using to communicate with c[i].

In the message modification action, the adversary randomly modifies a message in transit. This can occur in three cases. In the first case, server s[j]'s hellos message is modified, moving the protocol from safe state S.2 to error state U.0. When client c[i] receives the message and checks the integrity of the certificate, it detects the modification and discards the message, moving the protocol to unsafe state U.5. This causes c[i] to time out and return to safe state S.0. In the second case, c[i]'s finishc message is modified in transit, moving the protocol from safe state S.3 to error state U.2. When server s[j] receives the modified message and checks the integrity of its signature, it detects the modification and discards the message, moving the protocol to unsafe state U.6. This causes c[i] to time out and resend its finishc message, returning the protocol to safe state S.3. In the third case, s[j]'s finishs message is modified, moving the protocol from safe state S.4 to unsafe state U.4. When c[i] receives the message and checks its signature, it detects that the message is invalid and discards it, again moving the protocol to state U.6.

In the message replay action, the adversary replaces a message in transit with one it sent earlier. This can occur in three cases. In the first case, server s[j]'s hellos message is replaced with a replayed one that contains a certificate which has already expired. This causes the protocol to move from safe state S.2 to error state U.1. When client c[i] receives the message and checks the integrity of the certificate, it detects that the certificate has expired and discards the message, moving the protocol to unsafe state U.5. As before, this causes c[i] to time out and return to safe state S.0.

In the second case, c[i]'s finishc message is replaced with a previous one, moving the protocol from safe state S.3 to error state U.3. This attack fails since the secret, which is randomly generated in every session, is used in calculating the message's signature. When server s[j] receives the replayed message, it has no way to detect that the message has been replayed, so it responds with a finishs message by executing action s.1. This moves the protocol to U.4. When client c[i] receives the finishs message and checks the signature, it detects that the message was sent in response to a replayed helloc message and discards it, moving the protocol to U.6. Finally, c[i] times out and resends its finishc message, returning the protocol to safe state S.3.

In the third case, s[j]'s finishs message is replaced with an earlier one, moving the protocol from safe state S.4 to unsafe state U.4. Again, the attack fails since the random secret is used in calculating the message's signature. When c[i] receives the message and checks its signature, it detects that the message is invalid and discards it, again moving the protocol to state U.6.

To demonstrate that our SSL specification is secure against the adversary, we show that it satisfies the two conditions of convergence and protection discussed in section 3.

The convergence condition is satisfied since any computation whose first state is U.0, U.1, U.2, U.3, U.4, U.5, or U.6 includes safe state S.0 or S.3, as shown in figure 10.

To show that the protection condition is satisfied, we examine the protocol's critical variables. First, we examine c[i]'s critical variables:

- `certs` **: array** `[0..n-1]` **of integer,**
- `secret, key` **: array** `[0..n-1]` **of integer,**
- `esecret, emsg` **: array** `[0..n-1]` **of integer,**

Variables certs[j], secret[j], key[j], esecret[j], and emsg[j] can only be updated by action c.1, in which c[i] receives a hellos message from server s[j]. Assume the protocol starts in an unsafe state and the hellos message is modified or replayed. When c[i] receives the message, it first verifies the integrity of the certificate and verifies that the sender of the message matches the certificates owner. If the certificate is found to be invalid, c[i] discards the message, without updating any of its critical variables.

Now we examine s[j]'s only critical variable:

- `key` **: array** `[0..m-1]` **of integer,**

Variable key[i] can only be updated by action s.1, in which s[j] receives a finishc message from client c[i]. Again, assume the protocol starts in an unsafe state and the finishc message is modified or replayed. When s[j] receives the message, it verifies the integrity of the message by checking its signature. If the message is invalid, s[j] discards the message, without updating key[i].

10 Concluding Remarks

Our contributions in this paper are three-fold. First, in our formal specifications of Millicent and Micropayments, we introduced changes and simplifications that improve the protocols. For instance, with Millicent, we recognized that a customer needs only to have a single scrip for each vendor. The result is that there is virtually no requirement for the customer's computer to store large amounts of scrip, and the protocol never has to deal with combining scrip. We made similar simplifying assumptions in our specification of SSL.

Secondly, our specifications help expose the fundamental differences between the two digital cash protocols. While Millicent relies exclusively on hash functions to ensure message integrity, Micropayments uses public-key operations. Though this makes Micropayments slower and more resource-intensive, it is necessary for "universal acceptance," so that customers can immediately start buying from any Micropayments vendor. Millicent, on the other hand, is probably the more efficient of the two; however, scrip is vendor-specific.

Most importantly, we formally verified each protocol's security against an adversary capable of forgery, modification, and replay actions. In doing so, we outlined a method for verifying the security of protocols specified in the Abstract Protocol notation.

References

[1] A. O. Freier, P. Karlton, P. C. Kocher, *"The SSL Protocol Version 3.0"*, Internet-Draft, Nov. 1996.

[2] S. Glassman, M. S. Manasse, M. Abadi, P. Gauthier, P. Sobalvarro, "The MilliCent Protocol for Inexpensive Electronic Commerce", *Fourth International World Wide Web Conference*, Dec. 1995.

[3] M. G. Gouda, *Elements of Network Protocol Design*, John Wiley & Sons, New York, NY, 1998.

[4] M. G. Gouda, C.-T. Huang, A. Arora, "On the Security and Vulnerability of PING", *Fifth International Workshop on Self-Stabilizing Systems*, October 2001.

[5] A. Herzberg and H. Yochai, "MiniPay: Charging per Click on the Web", *Sixth International World Wide Web Conference*, Apr. 1997.

[6] R. Housley, W. Ford, W. Polk, D. Solo, *"Internet X.509 Public Key Infrastructure Certificate and CRL Profile"*, RFC 2459, Jan. 1999.

[7] B. Kaliski, J. Staddon, *"PKCS #1: RSA Cryptography Specifications Version 2.0"*, RFC 2437, Oct. 1998

[8] D. F. Lee, M. G. Gouda, "Specification and Verification of Secure Electronic Transactions (SET)", May 2001.

[9] M. S. Manasse, "The MilliCent Protocols for Electronic Commerce", *First USENIX workshop on Electronic Commerce*, July 1995.

[10] E. Rescorla, A. Schiffman, *"The Secure HyperText Transfer Protocol"*, RFC 2660, Aug. 1999.

[11] R. L. Rivest, *"The MD5 Message-Digest Algorithm"*, RFC 1321, Apr. 1992.

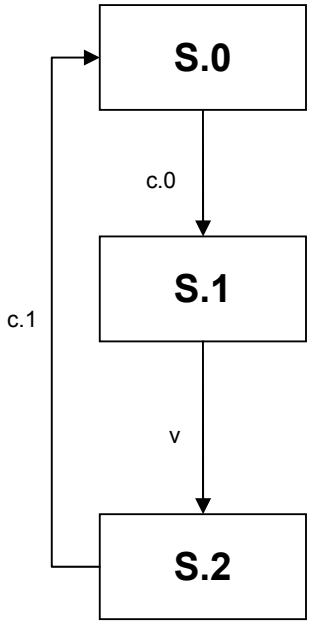
[12] R. L. Rivest, A. Shamir, "PayWord and MicroMint-Two Simple Micropayment Schemes", *Proceedings of 1996 International Workshop on Security Protocols*, 1996.

[13] SET Secure Electronic Transaction LLC, SET Secure Electronic Transaction Specification; Book 1: Business Description, May 1997.

[14] P. Wayner, *Digital Cash*, Academic Press, London, 1997.

Appendix A: Details of Millicent Verification

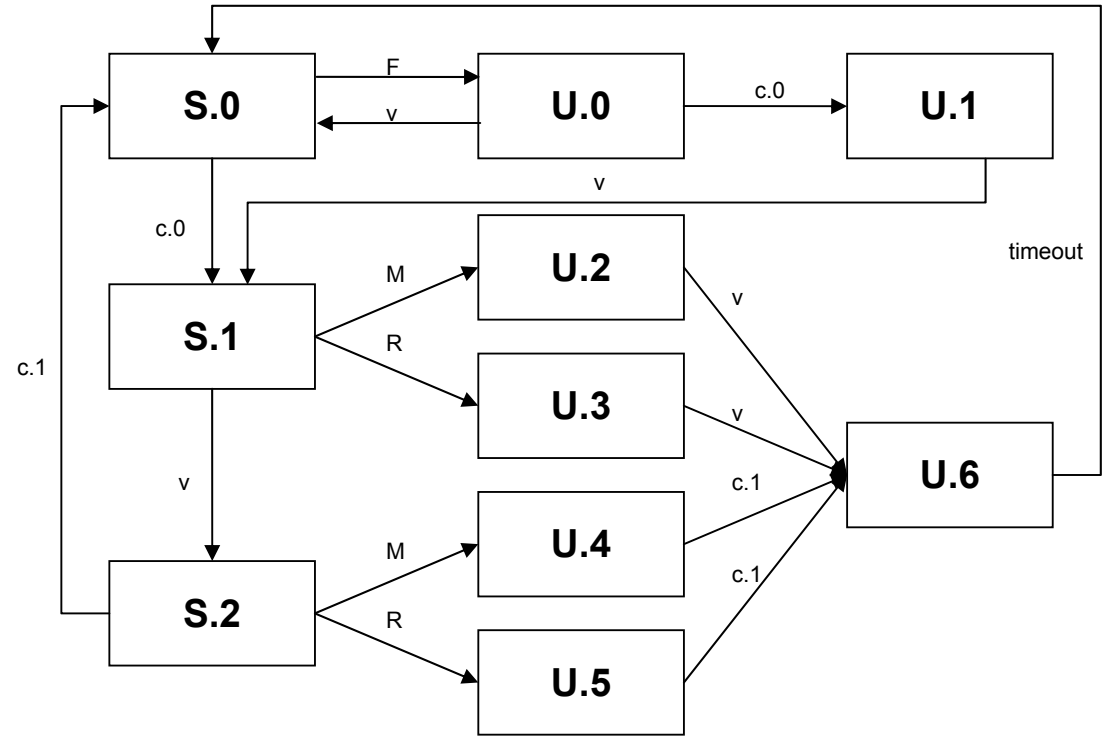
Figure 12 shows the formal details of the state transition diagram in figure 2.




```
S.0 = I ^ (ready[j] = true  in c[i]) ^
          (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {})
S.1 = I ^ (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {rqst(i, j, b, c, d, e, f)} ^ ch.v[j].c[i] = {})
          where b = val[j]   in c[i] ^
                c = seq[j]   in c[i] ^
                d = stamp[j] in c[i] ^
                e = MD(i | j | b | c | d | f | sc[j] in c[i]) ^
                f = newval   in c[i]
S.2 = I ^ (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {scrp(i, j, b, c, d, e)})
          where b = val[i]   in v[j] ^
                c = seq[i]   in v[j] ^
                d = stamp[i] in v[j] ^
                e = MD(i | j | b | c | d | sc[i] in v[j])
where
I = (seq[j]   in c[i] = seq[i]   in v[j]       v
     seq[j]   in c[i] = seq[i]-1 in v[j])
  ^ (seq[j]   in c[i] = seq[i]   in v[j]       =>
     val[j]   in c[i] = val[i]   in c[j])
  ^ (seq[j]   in c[i] = seq[i]   in v[j]       =>
     stamp[j] in c[i] = stamp[i] in v[j])
```

Figure 12. Detailed state transition diagram for Millicent

Figure 13 shows the formal details of the state transition diagram in figure 3.




```
U.0     = I ^ (ready[j] = true  in c[i]) ^
              (ch.c[i].v[j] = {rqst(i, j, b, c, d, e, f)} ^ ch.v[j].c[i] = {})
              where d ≠ MD(i | j | b | c | scv) ^
                    e = MD(i | j | b | c | d | f | sc[j] in c[i])
U.1     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {rqst(i, j, b, c, d, e, f),
                 rqst(i', j', b', c', d', e', f')} ^ ch.v[j].c[i] = {})
              where d ≠ MD(i | j | b | c | scv) ^
                    e = MD(i | j | b | c | d | f | sc[j] in c[i])
U.2     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {rqst(i, j, b, c, d, e, f)} ^ ch.v[j].c[i] = {})
              where e ≠ MD(i | j | b | c | d | f | sc[j] in c[i])
U.3     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {rqst(i, j, b, c, d, e, f)} ^ ch.v[j].c[i] = {})
              where c ≠ seq[j] in c[i] ^
                    c ≠ seq[i] in v[j] ^
                    e = MD(i | j | b | c | d | f | sc[j] in c[i])
U.4     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {scrp(i, j, b, c, d, e)})
              where e ≠ MD(i | j | b | c | d | sc[i] in v[j])
U.5     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {scrp(i, j, b, c, d, e)})
              where c ≠ seq[i] in v[j] ^
                    e = MD(i | j | b | c | d | sc[i] in v[j])
U.6     = I ^ (ready[j] = false in c[i]) ^
              (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {})
where
 I is as defined in Figure 12.
```

Figure 13. Detailed state transition diagram for Millicent

To complete the formal details of the proof, we formally define each of the adversary actions discussed in section 5.

Scrip Forgery

In this attack, the adversary creates its own scrip and sends it with a rqst message to v[j]. A valid rqst message looks like this:

```
rqst(i, j, b, c, d, e, f)
 where b = val[j]   in c[i] ^
       c = seq[j]   in c[i] ^
       d = stamp[j] in c[i] ^
       e = MD(i | j | b | c | d | f | sc[j] in c[i]) ^
       f = newval   in c[i]
```

The forged rqst message is of this format:

```
rqst(i, j, b', c', d', e', f')
 where d' ≠ MD(i | j | b' | c' | scv) ^
       e' = MD(i | j | b' | c' | d' | f' | sc[j] in c[i])
```

When the adversary sends its forged rqst message, the protocol moves from safe state S.0 to error state U.0. From here, two things can happen. In the first case, the vendor receives the rqst by executing action v and detects that the scrip has an invalid stamp. Hence, it throws the message away, and the protocol returns to safe state S.0. In the second case, the customer c[i] sends its own valid rqst message, moving the protocol to unsafe state U.1. When the vendor receives the adversary's invalid message, it discards it, bringing the protocol to safe state S.1.

Message Modification

There are two cases to consider. In the first, assume that process c[i] sends a valid rqst message to v[j]. This moves the protocol from state S.0 to S.1. The rqst message is of the following format:

```
rqst(i, j, b, c, d, e, f)
 where b = val[j]   in c[i] ^
       c = seq[j]   in c[i] ^
       d = stamp[j] in c[i] ^
       e = MD(i | j | b | c | d | f | sc[j] in c[i]) ^
       f = newval   in c[i]
```

When the adversary modifies the message, the protocol moves to error state U.2. We define the modified rqst to be:

```
rqst(i, j, b', c', d', e', f')
 where e' ≠ MD(i | j | b' | c' | d' | f' | sc[j] in c[i])
```

When v[j] receives the modified message, it detects that e', the signature, is invalid, and it throws the rqst away. This moves the protocol to unsafe state U.6. c[i] then times out and resends the rqst, causing the protocol to move back to S.0.

Now we consider what happens when the scrp message is modified. This is analogous to the case described above. Assume that v[j] sends a valid scrp message to c[i]. This moves the protocol from state S.1 to S.2. The scrp message is of the following format:

```
scrp(i, j, b, c, d, e)
 where b = val[i]   in v[j] ^
       c = seq[i]   in v[j] ^
       d = stamp[i] in v[j] ^
       e = MD(i | j | b | c | d | sc[i] in v[j])
```

When the adversary modifies the message, the protocol moves to error state U.4. The modified scrp message is defined as:

```
scrp(i, j, b', c', d', e')
 where e' ≠ MD(i | j | b' | c' | d' | sc[i] in v[j])
```

When c[i] receives the modified message, it will detect the invalid signature and throw the scrp away, moving the protocol to unsafe state U.6 again. From here, c[i] times out, and the protocol returns to safe state S.0.

Message Replay

Again, there are two cases to consider. First, we examine how vendors deal with replay attacks. Assume that process c[i] sends a valid rqst message to v[j]. This moves the protocol from state S.0 to S.1. Recall that the rqst message is of the following format:

```
rqst(i, j, b, c, d, e, f)
 where b = val[j]   in c[i] ^
       c = seq[j]   in c[i] ^
       d = stamp[j] in c[i] ^
       e = MD(i | j | b | c | d | f | sc[j] in c[i]) ^
       f = newval   in c[i]
```

When the adversary replaces this valid rqst message with a replayed one, the protocol moves to error state U.3. The replayed rqst message is defined to be:

```
rqst(i, j, b', c', d', e', f')
 where c' ≠ seq[j] in c[i] ^
       c' ≠ seq[i] in v[j] ^
       e  = MD(i | j | b | c | d | f | sc[j] in c[i])
```

When v[j] receives the replayed message, it detects that the sequence number is invalid and throws the rqst away. This moves the protocol again to unsafe state U.6. c[i] then times out and resends the rqst, causing the protocol to move back to safe state S.0.

The customer also remembers, for each vendor, the sequence number of its current scrip. For instance, imagine that c[i] sends a request with a scrip with serial number n. When c[i] receives the reply, it expects to see sequence number (n+1) in the new scrip.

Assume that v[j] sends a valid scrp message to c[i]. This moves the protocol from state S.1 to S.2. The scrp message is of the following format:

```
scrp(i, j, b, c, d, e)
```

```
 where b = val[i]   in v[j] ^
       c = seq[i]   in v[j] ^
       d = stamp[i] in v[j] ^
       e = MD(i | j | b | c | d | sc[i] in v[j])
```

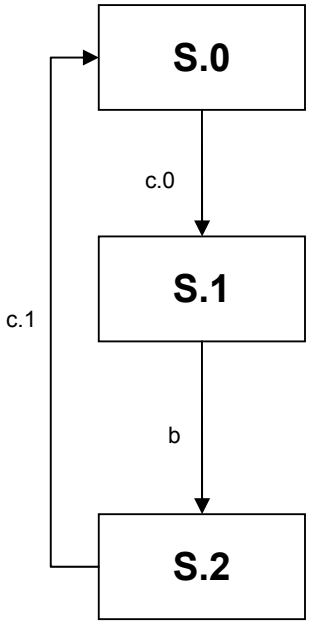
When the adversary replays an old scrp message, the protocol moves to error state U.5. The replayed scrp is of this format:

```
scrp(i, j, b′, c′, d′, e′)
 where c ≠ seq[i] in v[j] ^
       e = MD(i | j | b | c | d | sc[i] in v[j])
```

When c[i] receives the replayed message, it detects the invalid sequence number and throws the scrp away, moving the protocol to unsafe state U.6.

Appendix B: Details of Micropayments Verification

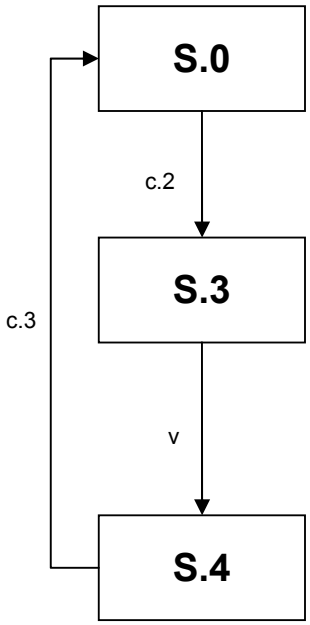
Figure 14 shows the formal details of the state transition diagram in figure 5.




```
S.0 = (ready[n] = true  in c[i]) ^ (ready[j] = true  in c[i]) ^
      (ch.c[i].b = {} ^ ch.b.c[i] = {} ^ ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {})
S.1 = (ready[n] = false in c[i]) ^
      (ch.c[i].b = {certrq(order')} ^ ch.b.c[i] = {})
      where (i', date', bal') := DCR(bkc[i], order') ^
             i'    = i ^
             date' = date
S.2 = (ready[n] = false in c[i]) ^
      (ch.c[i].b = {} ^ ch.b.c[i] = {certrp(cert')})
      where (i', bk', date', lim) := DCR(bkb, cert')
             i'    = i ^
             bk'   = bk in c[i] ^
             date' = date
```

Figure 14. Detailed state transition diagram for Micropayments – customer/broker

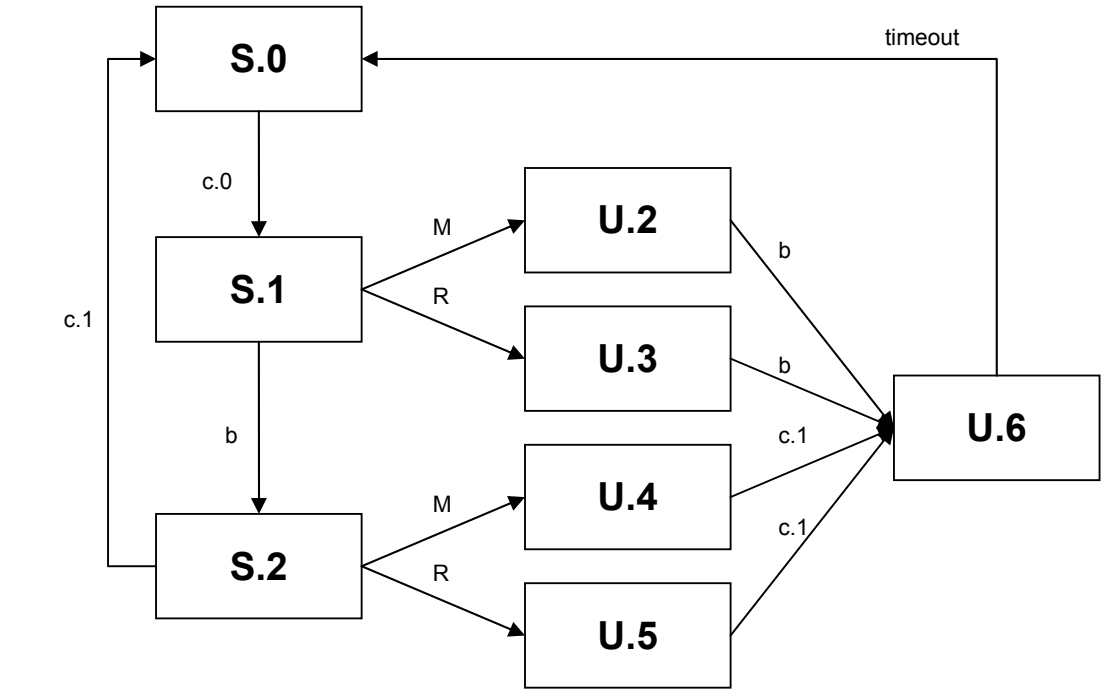
Figure 15 shows the formal details of the state transition diagram in figure 6.




```
S.3 = (ready[j] = false in c[i]) ^
      (ch.c[i].v[j] = {buyrq(cert', order')} ^ ch.v[j].c[i] = {})
      where (i', bkc', cdate', lim) := DCR(bkb, cert') ^
            (j', date', cost')      := DCR(bkc', order') ^
             i'     = i ^
             j'     = j ^
             cdate' = date ^
             date'  = date
S.4 = (ready[j] = false in c[i]) ^
      (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {buyrp(order')})
      where (i', j', date', cost') := DCR(bkv[j], order') ^
             i'    = i ^
             j'    = j ^
             date' = date
```

Figure 15. Detailed state transition diagram for Micropayments – customer/vendor

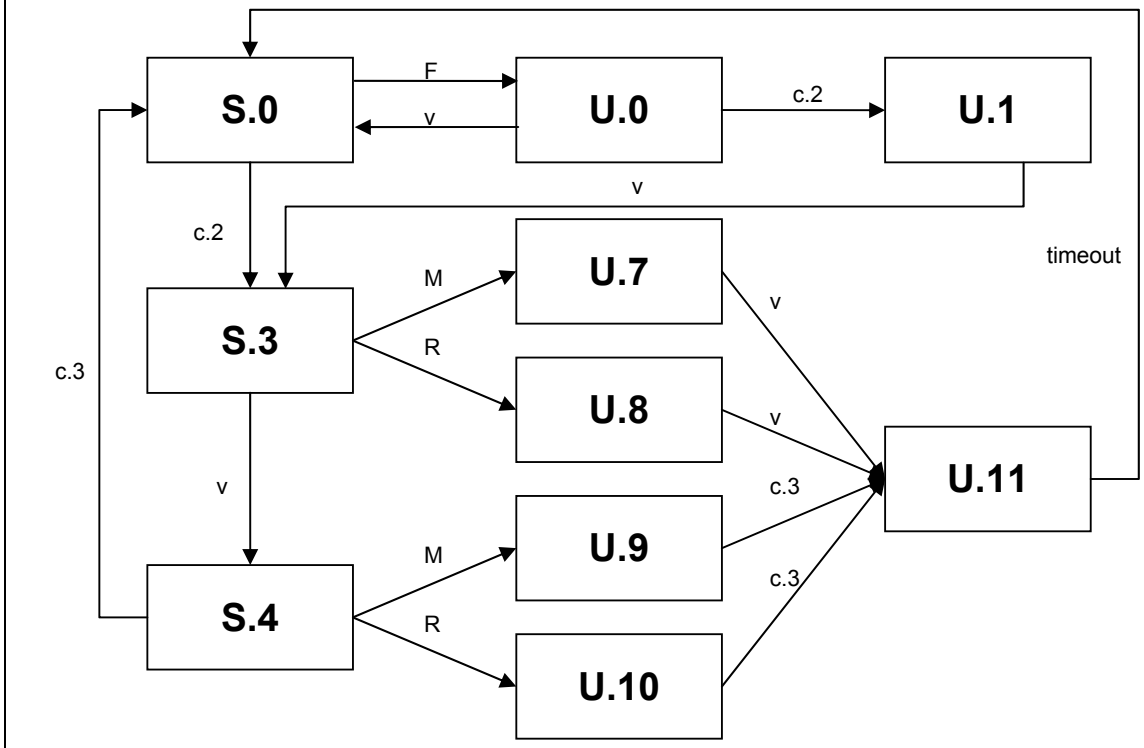
Figure 16 shows the formal details of the state transition diagram in figure 7.




```
U.2     = (ready[n] = false in c[i]) ^
          (ch.c[i].b = {certrq(order')} ^ ch.b.c[i] = {})
          where (i', date', bal') := DCR(bkc[i], order') ^
                (i'    ≠ i v
                 date' ≠ date)
U.3     = (ready[n] = false in c[i]) ^
          (ch.c[i].b = {certrq(order')} ^ ch.b.c[i] = {})
          where (i', date', bal') := DCR(bkc[i], order') ^
                 i'    = i ^
                 date' ≠ date
U.4     = (ready[n] = false in c[i]) ^
          (ch.c[i].b = {} ^ ch.b.c[i] = {certrp(cert')})
          where (i', bk', date', lim) := DCR(bkb, cert') ^
                (i'    ≠ i v
                 bk'   ≠ bk in c[i] v
                 date' ≠ date)
U.5     = (ready[n] = false in c[i]) ^
          (ch.c[i].b = {} ^ ch.b.c[i] = {certrp(cert')})
          where (i', bk', date', lim) := DCR(bkb, cert') ^
                 i'    = i ^
                 bk'   = bk in c[i] ^
                 date' ≠ date
U.6     = (ready[n] = false in c[i]) ^
          (ch.c[i].b = {} ^ ch.b.c[i] = {})
```

Figure 16. Detailed state transition diagram for Micropayments (customer/broker, with adversary)

Figure 17 shows the formal details of the state transition diagram in figure 8.




```
U.0     = (ready[n] = true  in c[i]) ^
          (ch.c[i].v[j] = {buyrq(cert', order')} ^ ch.v[j].c[i] = {})
          where (i', bkc', cdate', lim) := DCR(bkb, cert') ^
                (i'     ≠ i v
                 bkc'   ≠ bk in c[i] v
                 cdate' ≠ date)
U.1     = (ready[n] = false in c[i]) ^
          (ch.c[i].v[j] = {buyrq(cert', order'),
                           buyrq(cert'', order'')} ^ ch.v[j].c[i] = {})
          where (i', bkc', cdate', lim) := DCR(bkb, cert') ^
                (i'     ≠ i v
                 bkc'   ≠ bk in c[i] v
                 cdate' ≠ date)
U.7     = (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {buyrq(cert', order')} ^ ch.v[j].c[i] = {})
          where (i', bkc', cdate', lim) := DCR(bkb, cert') ^
                (j', date', cost')      := DCR(bkc', order') ^
                (i'     ≠ i v
                 bkc'   ≠ bk in c[i] v
                 cdate' ≠ date v
                 j'     ≠ j v
                 date'  ≠ date)
U.8     = (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {buyrq(cert', order')} ^ ch.v[j].c[i] = {})
          where (i', date', bal') := DCR(bkc[i], order') ^
                 i'     = i ^
                 bkc'   = bk in c[i] ^
                 cdate' ≠ date ^
                 j'     = j ^
                 date'  ≠ date ^
U.9     = (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {buyrp(order')})
          where (i', j', date', cost') := DCR(bkv[j], order') ^
```

```
                (i'    ≠ i v
                 j'    ≠ j v
                 date' ≠ date)
U.10    = (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {buyrp(order')})
          where (i', j', date', cost') := DCR(bkv[j], order') ^
                 i'    = i ^
                 j'    = j ^
                 date' ≠ date
U.11    = (ready[j] = false in c[i]) ^
          (ch.c[i].v[j] = {} ^ ch.v[j].c[i] = {})
```

Figure 17. State transition diagram for Micropayments (customer/vendor, with adversary)

To complete the formal details of the proof, we formally define each of the adversary actions discussed in section 7.

Certificate Forgery

In this attack, the adversary creates its own certificate and sends it in a buyrq message. (Recall that buyrq has two fields: cert, and order.) We assume that the adversary is colluding with the customer, c[i], so that it has access to c[i]'s private key. This enables the adversary to create a valid order.

A valid buyrq message is of the following format:

```
buyrq(cert', order')
 where (b, c, d, e) := DCR(bkb, cert') ^
       (f, g, h)    := DCR(bkc', order') ^
        b = i ^
        c = bk in c[i] ^
        d = date ^
        e = lim in b ^
        f = j ^
        g = date ^
        h = cost in c[i]
```

Now assume that the adversary sends a buyrq message with a forged certificate (moving the protocol from safe state S.0 to error state U.0). The invalid message looks like this:

```
buyrq(cert'', order'')
 where (b', c', d', e') := DCR(bkb, cert'') ^
       (f', g', h')     := DCR(bkc', order'') ^
       (b ≠ i v
        c ≠ bk in c[i] v
        d ≠ date) ^
        f' = f ^
        g' = g ^
        h' = h)
```

From here, two things can happen. In the first case, v[j] receives the buyrq, and detects that the included certificate is invalid; thus, v[j] discards the message and brings the protocol back to S.0. In the second case, c[i] sends its own, valid, buyrq message, moving the protocol to unsafe state U.1. Once the vendor v[j] receives the adversary's invalid buyrq message, the protocol goes to safe state S.3.

Message Modification

Since the protocol has four messages, there are four cases to consider:

1. c[i] sends a certrq message to b, moving the protocol from state S.0 to S.1. The original valid certrq message is of this format:

```
certrq(order')
  where (b, c, d) := DCR(bkc[i], order') ^
        b  = i ^
        c  = date ^
        d  = bal in c[i]
```

After the adversary modifies the message (moving the protocol from S.1 to error state U.2), it looks like this:

```
certrq(order'')
  where (b', c', d') := DCR(bkc[i], order'') ^
        (b' ≠ b v
         c' ≠ c)
```

When b receives the message, it detects that it was modified (when it checks the i' and date' fields) and throws the request away, moving the protocol to unsafe state U.6. This causes c[i] to timeout, moving the protocol back to safe state S.0.

2. b sends a certrp message to c[i], moving the protocol from state S.1 to S.2. The original certrp message is of this format:

```
certrp(cert')
   where (b, c, d, e) := DCR(bkb, cert') ^
         b = i ^
         c = bk in c[i] ^
         d = date ^
         e = lim in b
```

After the adversary modifies the message (moving the protocol to error state U.4), it looks like this:

```
certrp(cert'')
   where (b', c', d', e') := DCR(bkb, cert'') ^
         (b' ≠ b v
          c' ≠ c v
          d' ≠ d)
```

When c[i] receives and discards the message, the protocol again goes to state U.6.

3. c[i] sends a buyrq message to v[j], moving the protocol from state S.0 to S.3. The original buyrq message looks like this:

```
 buyrq(cert', order')
   where (b, c, d, e) := DCR(bkb, cert') ^
         (f, g, h)    := DCR(bkc', order') ^
          b = i ^
          c = bk in c[i] ^
          d = date ^
          e = lim in b ^
          f = j ^
          g = date ^
          h = cost in c[i]
```

After the adversary modifies the message (moving the protocol from S.3 to error state U.7), it looks like this:

```
buyrq(cert', order')
  where (b', c', d', e') := DCR(bkb, cert') ^
        (f', g', h')     := DCR(bkc', order') ^
        (b' ≠ b v
         d' ≠ d v
         f' ≠ f v
         g' ≠ g)
```

When v[j] receives the message, it detects that it was modified and discards the message, moving the protocol to U.11. This causes c[i] to timeout, moving the protocol back to safe state S.0.

4. v[j] sends a buyrp message to c[i], moving the protocol from state S.3 to S.4. The original buyrp message looks like this:

```
buyrp(order')
  where (b, c, d, e) := DCR(bkv[j], order') ^
        b = i ^
        c = j ^
        d = date ^
        e = cost in v[j]
```

After the adversary modifies the message (moving the protocol from S.4 to U.9), it looks like this:

```
buyrp(order'')
  where (b', c', d', e') := DCR(bkv[j], order'') ^
        (b' ≠ b v
         c' ≠ c v
         d' ≠ d)
```

When c[i] receives the message, it discards it, moving the protocol to U.11 again.

Message Replay

In the message replay action, the adversary replaces a valid message at the head of a channel with a message sent in a previous session (before the current day). As with modification, there are four cases to consider.

1. c[i] sends a certrq message to b, moving the protocol from state S.0 to S.1. Recall from above that the valid certrq message is of this format:

```
certrq(order')
  where (b, c, d) := DCR(bkc[i], order') ^
        b  = i ^
        c  = date ^
        d  = bal in c[i]
```

After the adversary replays an old certrq (moving the protocol from S.1 to error state U.3), it looks like this:

```
certrq(order'')
  where (b', c', d') := DCR(bkc[i], order'') ^
        b' = b ^
```

```
c′ ≠ c
```

When b receives the message, it detects that it was replayed (when it checks the date' field) and discards the message, moving the protocol to U.6. This causes c[i] to timeout, moving the protocol back to safe state S.0.

2. b sends a certrp message to c[i], moving the protocol from state S.1 to S.2. The original certrp message is of this format:

```
certrp(cert′)
   where (b, c, d, e) := DCR(bkb, cert′) ^
          b = i ^
          c = bk in c[i] ^
          d = date ^
          e = lim in b
```

After the adversary replays an old certrp (moving the protocol from S.2 to error state U.5), it looks like this:

```
certrp(cert′′)
   where (b′, c′, d′, e′) := DCR(bkb, cert′′) ^
          b′ = b ^
          c′ = c ^
          d′ ≠ d
```

When c[i] receives the message, it throws it away, moving the protocol to U.6 again.

3. c[i] sends a buyrq message to v[j], moving the protocol from state S.0 to S.3. Recall that the original buyrq message looks like this:

```
buyrq(cert′, order′)
  where (b, c, d, e) := DCR(bkb, cert′) ^
        (f, g, h)    := DCR(bkc′, order′) ^
         b = i ^
         c = bk in c[i] ^
         d = date ^
         e = lim in b ^
         f = j ^
         g = date ^
         h = cost in c[i]
```

After the adversary replays an old buyrq (moving the protocol from S.3 to U.8), it looks like this:

```
buyrq(cert′, order′)
  where (b′, c′, d′, e′) := DCR(bkb, cert′) ^
        (f′, g′, h′)     := DCR(bkc′, order′) ^
         b′ = b ^
         d′ ≠ d ^
         f′ = f ^
         g′ ≠ g
```

When v[j] receives the message, it detects that it was modified and discards the message, moving the protocol to U.11. This causes c[i] to timeout, moving the protocol back to S.0.

4. v[j] sends a buyrp message to c[i], moving the protocol from state S.3 to S.4. The original buyrp message looks like this:

```
buyrp(order′)
  where (b, c, d, e) := DCR(bkv[j], order′) ^
```

```
b = i ^
c = j ^
d = date ^
e = cost in v[j]
```

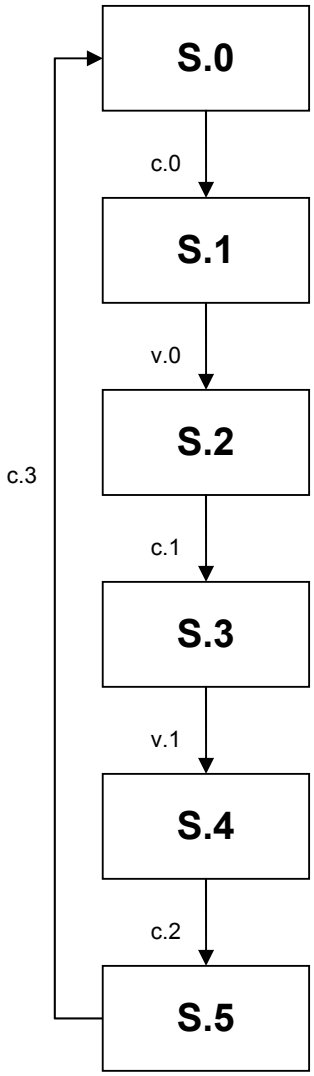
After the adversary modifies the message (moving the protocol from S.4 to U.10), it looks like this:

```
buyrp(order'')
  where (b', c', d', e') := DCR(bkv[j], order'') ^
         b' = b ^
         c' = c ^
         d' ≠ d
```

When c[i] receives the message, it discards it, moving the protocol to S.0 again.

Appendix C: Details of SSL Verification

Figure 18 shows the formal details of the state transition diagram in figure 10.

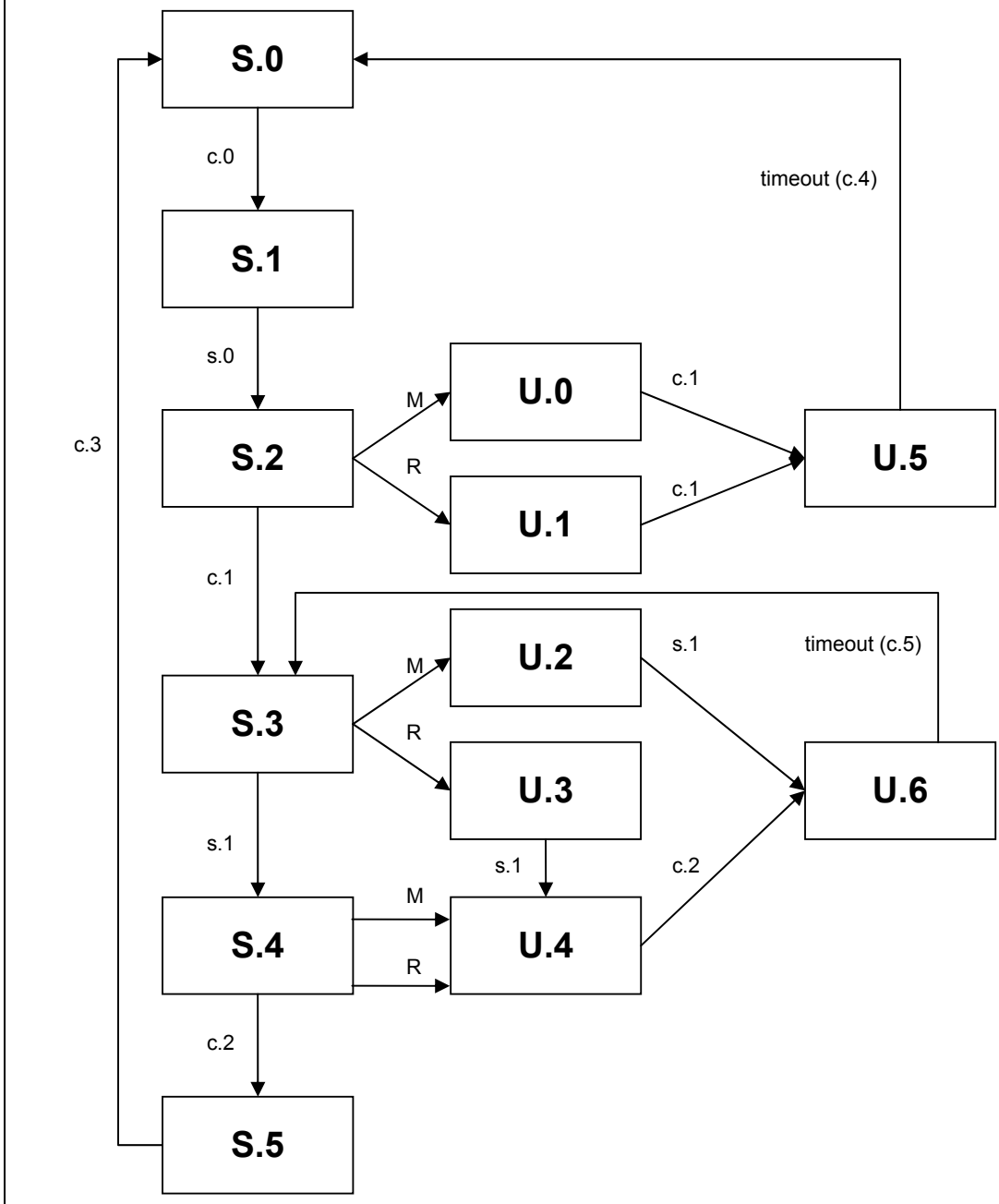



```
S.0 = (st[j] = 0 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {})
S.1 = (st[j] = 1 in c[i]) ^
      (ch.c[i].s[j] = {helloc} ^ ch.s[j].c[i] = {})
S.2 = (st[j] = 1 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {hellos(certs')})
      where (j', bks', date') := DCR(bkca, certs') ^
             j'    =  j ^
             bks'  =  bk in s[j] ^
             date' >= date
S.3 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {finishc(esecret', emsg')} ^ ch.s[j].c[i] = {})
      where (secret') := DCR(rk, esecret') ^ (msg') := DCR(key', emsg') ^
             secret'  =  secret[j] in c[i] ^
             key'     =  F(secret') ^
             msg'     =  G(i, j, certs[j], secret[j], esecret[j]) in c[i]
```

```
S.4 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {finishs(emsgs')})
      where (msgs') := DCR(key[j], emsgs') ^
            msgs' =  H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
S.5 = (st[j] = 3 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {})
```

Figure 18. Detailed state transition diagram for SSL

Figure 19 shows the formal details of the state transition diagram in figure 11.




```
U.0 = (st[j] = 1 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {hellos(certs')})
      where (j', bks', date') := DCR(bkca, certs') ^
```

```
            (j'    ≠  j ^
             bks'  ≠  bk in s[j] ^
             date' ≠  date in cert in s[j])
U.1 = (st[j] = 1 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {hellos(certs')})
      where (j', bks', date') := DCR(bkca, certs') ^
            (j'    =  j ^
             bks'  =  bk in s[j] ^
             date' <  date)
U.2 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {finishc(esecret', emsg')} ^ ch.s[j].c[i] = {})
      where (secret') := DCR(rk, esecret') ^ (msg') := DCR(key', emsg') ^
            (secret' ≠  secret[j] in c[i] v
             msg'    ≠  G(i, j, certs[j], secret[j], esecret[j]) in c[i]))
U.3 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {finishc(esecret', emsg')} ^ ch.s[j].c[i] = {})
      where (secret') := DCR(rk, esecret') ^ (msg') := DCR(key', emsg') ^
            (secret' ≠  secret[j] in c[i] ^
             msg'    ≠  G(i, j, certs[j], secret[j], esecret[j]) in c[i]))
U.4 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {finishs(emsgs')})
      where (msgs') := DCR(key[j], emsgs') ^
             msgs' ≠  H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
U.5 = (st[j] = 1 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {})
U.6 = (st[j] = 2 in c[i]) ^
      (ch.c[i].s[j] = {} ^ ch.s[j].c[i] = {})
```

Figure 19. Detailed state transition diagram for SSL (with adversary)

To complete the formal details of the proof, we formally define each of the adversary actions discussed in section 9.

Message Modification

There are three cases to consider:

1. s[j] sends a hellos message to c[i], moving the protocol from state S.1 to S.2. The original valid hellos message is of this format:

```
hellos(certs')
  where (b, c, d) := DCR(bkca, certs') ^
         b  =  j ^
         c  =  bk in s[j] ^
         d  >= date
```

After the adversary modifies the message (moving the protocol from S.2 to error state U.0), it looks like this:

```
hellos(certs'')
  where (b', c', d') := DCR(bkca, order'') ^
        (b' ≠  b ^
         c' ≠  c ^
         d' ≠  d)
```

When c[i] receives the message, it detects that it was modified and discards it, moving the protocol to unsafe state U.5. This causes c[i] to timeout, moving the protocol back to safe state S.0.

2. c[i] sends a finishc message to s[j], moving the protocol from state S.2 to S.3. The original finishc message is of this format:

```
finishc(esecret', emsg')
   where (b) := DCR(rk, esecret') ^ (d) := DCR(c, emsg') ^
          b = secret[j] in c[i] ^
          c = F(b) ^
          d = G(i, j, certs[j], secret[j], esecret[j]) in c[i]
```

After the adversary modifies the message (moving the protocol to error state U.2), it looks like this:

```
finishc(esecret'', emsg'')
   where (b') := DCR(rk, esecret'') ^ (d') := DCR(c', emsg'') ^
         (b' ≠ b v
          d' ≠ d)
```

When s[j] receives and discards the message, the protocol goes to state U.6. This causes c[i] to timeout, moving the protocol back to safe state S.3.

3. s[j] sends a finishs message to c[i], moving the protocol from state S.3 to S.4. The original finishs message looks like this:

```
finishs(emsgs')
  where (b) := DCR(key[j], emsgs') ^
         b  = H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

After the adversary modifies the message (moving the protocol from S.4 to error state U.4), it looks like this:

```
finishs(emsgs'')
  where (b') := DCR(key[j], emsgs'') ^
         b' ≠ H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

When c[i] receives the message, it detects that it was modified and discards the message, moving the protocol again to U.6.

Message Replay

As with modification, there are three cases to consider.

1. s[j] sends a hellos message to c[i], moving the protocol from state S.1 to S.2. Recall the original valid hellos message is of this format:

```
hellos(certs')
  where (b, c, d) := DCR(bkca, certs') ^
         b  =  j ^
         c  =  bk in s[j] ^
         d  >= date
```

When the adversary replaces this valid hellos message with a replayed one (which includes an expired certificate), the protocol moves to error state U.1. The replayed hellos message is defined to be:

```
hellos(certs'')
  where (b', c', d') := DCR(bkca, order'') ^
        (b' =  b ^
         c' =  c ^
```

```
        d' <  date)
```

When c[i] receives the message, it detects that it was replayed and discards it, moving the protocol to unsafe state U.5. This causes c[i] to timeout, moving the protocol back to safe state S.0.

2. c[i] sends a finishc message to s[j], moving the protocol from state S.2 to S.3. Recall the original finishc message is of this format:

```
finishc(esecret', emsg')
   where (b) := DCR(rk, esecret') ^ (d) := DCR(c, emsg') ^
          b = secret[j] in c[i] ^
          c = F(b) ^
          d = G(i, j, certs[j], secret[j], esecret[j]) in c[i]
```

When the adversary replaces this valid finishc message with a replayed one, the protocol moves to error state U.3. The replayed finishc message is defined to be:

```
finishc(esecret'', emsg'')
   where (b') := DCR(rk, esecret'') ^ (d') := DCR(c', emsg'') ^
         (b' ≠ b ^
          d' ≠ d)
```

When s[j] receives the message, it is unable to detect the attack, so it replies with a finishs message, moving the protocol to state U.4. Recall that a valid finishs message should be of this format:

```
finishs(emsgs')
  where (b) := DCR(key[j], emsgs') ^
         b = H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

However, the finishs message the server sends is of this format:

```
finishs(emsgs'')
  where (b') := DCR(key[j], emsgs'') ^
         b' ≠ H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

When c[i] receives the finishs message, it detects that the message is invalid and discards it, moving the protocol to state U.6. This causes c[i] to timeout, moving the protocol back to safe state S.3.

3. s[j] sends a finishs message to c[i], moving the protocol from state S.3 to S.4. The original finishs message looks like this:

```
finishs(emsgs')
  where (b) := DCR(key[j], emsgs') ^
         b  = H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

When the adversary replaces this valid finishs message with a replayed one, the protocol moves to error state U.4. The replayed finishs message is defined to be:

```
finishs(emsgs'')
  where (b') := DCR(key[j], emsgs'') ^
         b' ≠ H(i, j, certs[j], secret[j], esecret[j], emsg[j]) in c[i]
```

When c[i] receives the message, it detects that it was modified and discards the message, moving the protocol again to U.6.