Specification and Verification of the Payword* Protocols

Ali Lakhia†
lakhia@cs.utexas.edu

Department of Computer Sciences
University of Texas at Austin

May 19, 2000

Abstract

When many small payments need to be made by a buyer over a computer network, considerable computational overhead, per-transaction fees and delay make most schemes infeasible. Therefore, we devised a class of protocols based on the Payword micro-payment scheme. During the process of specifying the protocols using the Abstract Protocol notation[1], we discovered a way for an adversary to disrupt the exchange of payments. The protocol that we present is more secure and robust against attacks by any adversary that can lose, modify and replay messages. We also present a few variations to the Payword protocol and prove formally that the protocol is correct.

*This protocol is based on the paper by Ronald Rivest and Adi Shamir.

†This work was collaborated with Lok Yan Leung. This project would not have been possible without Dr. Gouda's guidance and supervision.

[1]Abstract Protocol notation of Mohamed G. Gouda.

Contents

1 Overview of the Payword Protocol

Frequently, a buyer must make numerous small purchases over a computer network. During these circumstances, both the buyer and the seller incur considerable computational overhead. This overhead comes from a number of public-key encryption operations that must take place so that the buyer and the seller can ensure that nobody else is pretending to be the other party. Also, the protocol needs to be sophisticated enough to detect modification of messages by somebody while in transit.

Most schemes in existence utilize the existing credit card infrastructure. While this form of payment serves well for large payments, it is impractical when the payments are small because each transaction typically involves a minimum fee of 20 cents.

This motivated us to design a network protocol to efficiently exchange cash between two processes through a computer network. This protocol would minimize the computational overhead and provide a secure and robust method to send payments.

1.1 Criteria

Let us define the criteria of the expected solution more precisely. We expect that our protocols should provide:

1. *Authentication*: The buyer and the seller must verify that the other party is really who he or she claims to be. That is, a third party is not impersonating either the buyer or the seller.

2. *Integrity of payments*: The seller is assured that the payment it received is valid and was not modified while in transit. Also, the payment could only have been sent by the buyer it previously authenticated.

3. *Integrity of receipt of payments*: The buyer is guaranteed that the seller received the payments it sent. Also, the payment ought to be the same amount as sent by the buyer.

1.2 Background

The Payword protocol, presented by Ronald Rivest and Adi Shamir in their paper "PayWord and MicroMint: Two simple micropayment schemes" suited our purpose. The payword protocol almost eliminated public-key encryptions by using hash functions instead.

The payword protocol, a credit based scheme, relies on a chain of hash values to represent payments. Since all payments are made on credit, the seller collects all payments till the end of a billing cycle. Then the seller gets reimbursed by a bank with a single transaction. This significantly reduces per-transaction fees. Also hash functions are about a 100 times faster than RSA signature verification and are considerably less computationally expensive than macro-payment schemes.

The protocol, as described by Rivest and Shamir, calls for a "chain of paywords" created by the buyer. The buyer would generate the chain such that:

$$\forall i : 0 \leq i < n : c[i+1] = H(c[i])$$

where H is a one-way hash function, c is an array of length $n+1$ containing the chain, and n is a constant.

A payword, an integer hash value, represents cash. Each payword could easily be verified by performing a one-way hash function on it a certain number of times and comparing with the previous payword.

In other words, a payment contains a payword and an integer representing the cumulative number of paywords sent. Figure 1 shows a typical exchange of payments between the buyer, p and the seller q. At any point in time, q has the previous payword that it uses to verify each new payword.

This protocol by Rivest and Shamir has two weaknesses. First, an adversary can discard a payment message from the network channel. The buyer, p, has no means of knowing if a payment sent was actually received by q.

The second weakness is less obvious. Let us elaborate on how an adversary can exploit the protocol. It is not unreasonable to assume that the adversary can figure out what the hash function, H is. When the adversary sees a payment in the channel from p to q, it can hash the payword a certain number of times. If the adversary decrements the second field in the payment by the same amount, the payment is a valid one.

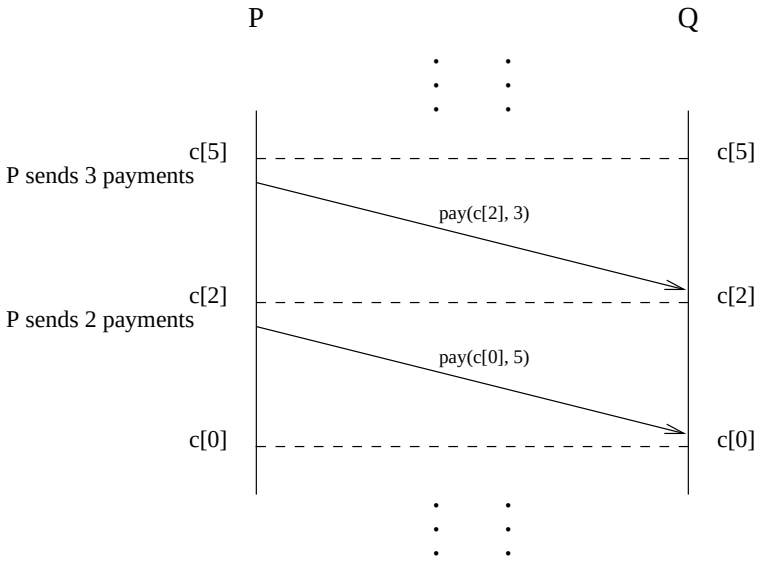

Figure 1: A typical exchange of cash between the buyer p and the seller q.

For example, consider a situation where the chain's length is 5. Both p and q have communicated $c[5]$ to each other. Now p wishes to send three paywords. This payment would be:

$$\text{p} \rightarrow \text{q} : \text{pay}(c[2], 3)$$

While in transit, an adversary hashes $c[2]$ once and replaces 3 with a 2. This new message is placed in the channel from p to q:

$$\text{p} \rightarrow \text{q} : \text{pay}(c[3], 2)$$

Note that $H(c[2]) = c[3]$.

When q receives this payment, it will apply the hash function on $c[3]$ twice. It will then compare the result with c[5]. Since $H^2(c[5])$ equals $c[3]$, q would accept this message. This error is illustrated in figure 2.

Our challenge was to fix the previous two errors. We had to do so without sacrificing efficiency—otherwise, that would defeat the purpose of having a light-weight micro-payment scheme.

In the next section, we look at a basic version of the payword protocol which gives us a better understanding of the payword protocol. In this version, we provide a solution to the first problem present in the original protocol. We also prove formally that the Basic Payword Protocol is correct. Then we discuss the Variable Payword Protocol and provide a fix for the second error.

1.3 Requirements and Assumptions

Before we look at any protocols, we state the assumptions we made during development and specification of the Payword protocols:

1. This protocol requires a one-way hash function, H, that is not distributive on concatenation:

   $H(x; y) \neq H(x); H(y)$ for arbitrary x and y

2. Finite number of communication errors occur.

3. An adversary will remove, modify or replay messages a finite number of times.

4. Both parties have agreed on the value of each payword.

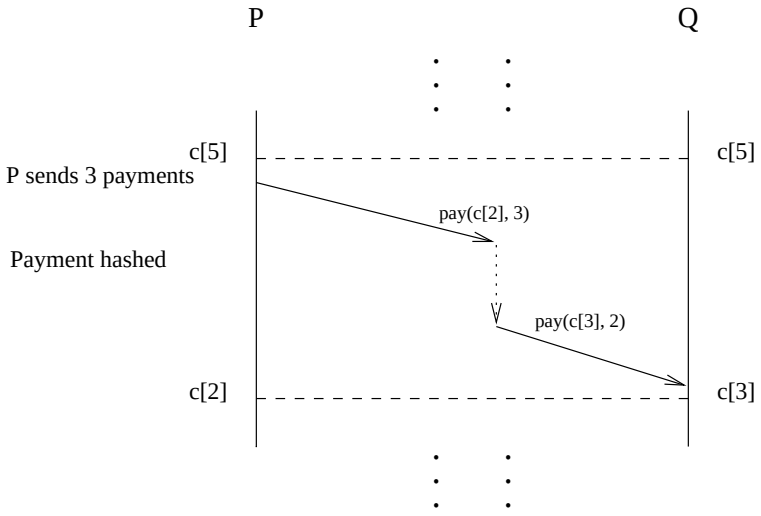

Figure 2: This picture illustrates how an adversary can disrupt communication between the buyer p and the seller q by hashing the first field and decrementing the second.

2 Development of the Basic Payword Protocol

The protocol consists of two processes p and q. Here, process p is the buyer and process q is the seller. In this protocol, the amount of each payment is fixed beforehand. The Variable Payword Protocol, that we present later, allows for amount of payments to vary.

As we discussed is section 1.2, this protocol does not provide protection for message loss. For example, if an adversary removed the payment from the channel, process p would still be under the impression that process q received the payment. Therefore, we need a mechanism so that q could acknowledge receipt of payments.

The easiest way to acknowledge payments is for q to have an acknowledgment chain. When p sends $c[n]$ to q, q would reply with $d[n]$ where d is q's acknowledgment chain. Likewise, when q receives each payment, it sends an acknowledgment from its acknowledgment chain.

When process p wants to make a purchase, it will send a request message to process q. Process q will verify the authenticity of the request and send a reply to confirm receipt of the request. Then, both processes can start exchanging cash.

2.1 Sending a Request

The exchange of payments is initiated by sending a request to process q. The request indicates p's willingness to pay for q's products and/or services. It also allows both parties to authenticate each other.

Before process p can send a request to initiate exchange of payments, it will generate the chain of paywords. First, process p sets the last payword, $c[0]$, to a random number. It is very important to pick this number carefully since the entire chain of paywords is derived from this number. Using the same number twice will result in two identical chains, making it easy for someone to predict future payments.

Then, process p applies the hash function n number of times to $c[0]$ and stores each hash value in the corresponding element in the chain. Thus, the last hash operation gives us $H^n(c[0])$ or $c[n]$. The resulting chain is shown in figure 3. The last element, $c[n]$, is known as the "root the chain." The root is sent encrypted to q within the request.

When process q receives the request, it would generate another chain as outlined above. That is, it would set its $d[0]$ to a random number, hash it n times and store each hash in the array d. Process q would then send a reply containing the root of its chain.

Let us first consider a simple attempt to exchange the roots of the chains. In this attempt, process p sends to process q the root of the chain of paywords encrypted by their shared key. Upon receiving the root,

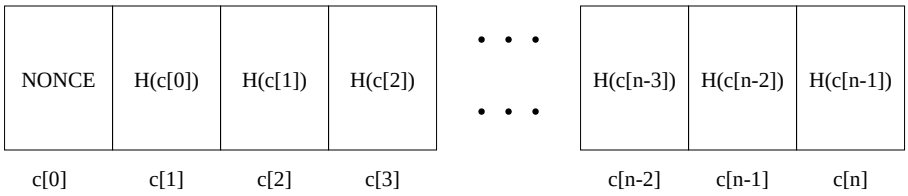

Figure 3: The payword chain, c, in process p is shown above.

process q does the same. This can be represented as follows:

$$\text{p} \rightarrow \text{q} : \text{S}\langle c[n]\rangle$$
$$\text{q} \rightarrow \text{p} : \text{S}\langle d[n]\rangle$$

There are two problems with this approach. First, an adversary could intercept a message from p to q, create another random message and put it in the channel from p to q. Process q would decrypt it using the shared key and use that as p's root. The adversary could do the same for the response from q to p. Thus, both processes would be unable to exchange cash because the hash function applied to the payment (or acknowledgment) would not equal the root of the chain. The other problem with the protocol is that the adversary could replay the above messages at another time and disrupt the communication between the two processes.

To fix these two problems, we must use unique identifiers every time the root of the chains need to be exchanged. A sequence number serves this purpose well. Both processes start with their sequence numbers set to a particular value. Process p concatenates this sequence number to the message and process q verifies it. Process p and q increment their sequence numbers by one when both are convinced that the other process has used the current sequence number. This exchange is shown below:

$$\text{p} \rightarrow \text{q} : \text{S}\langle c[n]; seq\rangle$$
$$\text{q} \rightarrow \text{p} : \text{S}\langle d[n]; seq\rangle$$

Note that the sequence numbers are encrypted also. This ensures that an adversary cannot create a random message that would be accepted by either processes because the sequence number would not match. Also, replaying old messages would be rejected because the new sequence number would be greater than the one used previously.

Also, it is nearly impossible for an adversary to modify the request such that the sequence number is valid. Since the hash function is not distributive over concatenation, modifying even the most significant bits of the message would cause the sequence number to change. Therefore, we can rest assured that the above exchange provides authentication.

2.2 Exchanging Cash

After the exchange of the two messages, both processes have authenticated each other. Now, process p can start making payments.

To send a payment, p uses the next unused payword from its chain and sends it unencrypted to q. Process q responds with an acknowledgment. The first payment to q is as follows:

$$\text{p} \rightarrow \text{q} : c[n-1]$$
$$\text{q} \rightarrow \text{p} : d[n-1]$$

When p receives the acknowledgment, p has confirmed that q received its previous payment because only process q could have generated the acknowledgment. Therefore, process p must wait for an acknowledgment before sending the next payment.

After the first payment has been sent, p can continue by sending $c[n-2]$ as the second payment, $c[n-3]$ as third and so on till $c[0]$. Similarly, q will acknowledgment with $d[n-2]$ for the second payment, $d[n-3$ for third and so on. Thus, a total of n exchanges can be made between p and q using this protocol.

Both processes can verify the payword or acknowledgment simply by applying H once to it and verifying that the result is the previous payword or acknowledgment. However, no one can anticipate what the next payment or acknowledgment will be because the hash function is a one-way function.

To start a new chain, process p must wait until it receives an acknowledgment from any pending payment. In other words, after receiving each acknowledgment, p has two choices. Firstly, it could generate a new chain and send a request message to q. The other choice is to continue with the current chain. However, if the chain is empty, p has no choice but to generate a new chain and send another request message. Thus, the whole process repeats.

3 Specification of the Basic Payword Protocol

The protocol specification is presented in Abstract Protocol notation, or $\mathcal{AP}$ notation. The $\mathcal{AP}$ notation enables us to define the protocol without ambiguities or unnecessary details.

3.1 Data structures

Both processes need an array of integers to store the chain, named c. Although p's chain and q's chain are both called c, the two chains are independent of each other. Variables seq and akn are used to indicate the next sequence number and the last integer hash value of the other process respectively. Process p also needs a variable to maintain the remaining length of its chain with rem. The variable pos is used by q to store its position in the acknowledgment chain and the number of payments received from p.

Other variables used by p are u and v which are used to temporarily store decrypted information from q. Similarly, q needs t and u to store data received from p. Process q also uses t to compute the chain.

In addition to the variables above, p can be in four states at any point in time. Thus, it needs a variable:

st : **set** $\{rp, rp, py, ak\}$

where rp indicates that p can generate a new chain and send a request message to q and py means p can send a payment. The states rp, and ak mean that p is waiting for a reply to the request, and waiting for acknowledgment for a payment respectively.

3.2 Abstract Protocol Specification

```
process p
    const sk, n
    var     st :              set {rq,rp,py,ak},          {init rq}
            c :               array [0..n] of integer,
            seq, akn, u, v :  integer                     {seq init 1}
            rem :             0..n,
    begin
            st = rq ⟶ st, c[0], rem := rp, NONCE, 0;
                      do (rem<n) →
                              c[rem+1], rem := H(c[rem]), rem+1
                      od;
                      send rqst (NCR(sk, (c[n]; seq))) to q

    []      rcv rply(u) from q ⟶
                      u, v := DCR(sk, u);
                      if st=rp ∧ v=seq → st, seq, akn := py, seq+1, u
                      [] st≠rp ∨ v≠seq → skip
                      fi

    []      st = py ⟶ st, rem := ak, rem-1;
                      send pay(c[rem]) to q;
```

```
    []      rcv ack(u) from q ⟶
                    if akn=H(u) ∧ rem>0 → st, akn := py, u
                    [] akn=H(u) → st := rq
                    [] akn≠H(u) → skip
                    fi

    []      timeout (#ch.p.q + #ch.q.p = 0) ∧ (st=ak ∨ st=rp) ⟶
                    if st=rp → send rqst (NCR(sk, (c[n]; seq))) to q
                    [] st=ak → send pay(c[rem]) to q;
                    [] st≠rp ∧ st≠ak → skip
                    fi
    end.


process q
    const sk, n
    var     c :             array [0..n] of integer,
            seq, akn, t, u : integer                    {seq init 1}
            pos:            integer
    begin
            rcv rqst(t) from p ⟶
                    t, u := DCR(sk, t);
                    if u=seq → seq, c[0], akn, t, pos := sq + 1, NONCE, t, 0, 0;
                                do (t<n) →
                                      c[t + 1], t := H(c[t]), t + 1
                                od;
                                send rply(NCR(sk, (c[n]; seq - 1))) to p;
                    [] u=seq - 1 → send rply(NCR(sk, (c[n]; seq - 1))) to p;
                    [] u≠seq ∧ u≠seq - 1 → skip
                    fi

    []      rcv pay(t) from p ⟶
                    if akn=H(t) → akn, pos := t, pos + 1;
                                  send ack(c[n - pos]) to p
                    [] akn≠H(t) → send ack(c[n - pos]) to p
                    fi
    end.
```

3.3 Discussion

Process p

Of the five actions in p, the first action occurs when p is in the request state. Here, p generates a new chain by first setting $c[0]$ to NONCE and then hashing n times. Note that NONCE returns an integer that is different from all the integers previously returned. This prevents creation of identical and predictable chains. Then, process p sends the root to q along with the sequence number. The state of p is changed to reflect that it is now waiting for a reply from q.

In the second action, p decrypts q's reply. Then, p checks if it was waiting for a reply from q and that the sequence number is the same as it was expecting. If either conditions fail, p discards the message. This implies that p will disregard replies if it was not expecting one and that q cannot change its chain unless p wishes to do so as well. On the other hand, if both conditions are satisfied, p stores the root in *akn*, increments *seq* and changes its state to *py* to indicate it is ready to make payments.

To make a payment, process p proceeds to the third action. This action is only executed if there is atleast one payword left in the chain ($rem > 0$). Here, process p changes state to reflect it is waiting for an acknowledgment, decrements rem by 1, and sends a payment using $c[rem]$ as the payword.

The fourth action handles acknowledgments. It hashes the acknowledgment once and compares with akn. If both are the same, p is assured that q received the previous payment and changes akn to the received acknowledgment. Otherwise, it discards the acknowledgment.

Note that p does not always go to the py state from the ak state. If the remaining payword chain is greater than a length of 0, p non-deterministically either goes back to the rq state or py state. On the other hand, if the length of the chain is 0, it will always go to the rq state.

This behaviour guarantees two things. First, if p changes state to py, there is atleast one payword remaining in the chain. Second, If process p goes to the rq state, that denotes that p is discarding the previous chain and starting all over. The payments made uptill this point are still valid, though.

The final action accounts for message loss. It timeouts if the channels between p and q are empty and if p is waiting for either a reply or an acknowledgment. It then simply resends the previous request or payword depending on its state.

Process q

Let us now examine process q. It has a total of two actions. The first action deals with a request from p. If process q receives the decrypted sequence number that it was expecting, it generates a new chain and replies to p. If the number is the previous one, it means that p did not receive the previous reply and q resends it without computing a new chain. If both cases are false, q discards the message.

The second action handles payments from p. Process q hashes the payword, t, once and compares it with the previous payword, stored in akn. If the comparison is true, q sends an acknowledgment. However, if the payword is the same as the previous payword received by q, this indicates that the previous acknowledgment was not received by p, and process q resends it without modifing its internal state.

Note that q uses pos to store the position in its chain. This variable also denotes the number of paywords received from p. Therefore, q increments pos everytime it gets a valid payword. Also note that q's chain is equal to the length of p's chain so it will never use up all of its chain before p.

4 Verification of the Basic Payword Protocol

The protocol that we presented, was derived from trial and error. However, that does not mean that the Basic Payword Protocol is correct. Therefore, in this section, we prove formally that the protocol is correct. We do this in three stages.

4.1 Correctness of the the Basic Payword Protocol

First, we prove that the protocol is correct, assuming that no communication errors take place and that an adversary does not try to disrupt the communication between the two processes in any way.

Let us examine the state of the protocol in its initial state[2], S.0:

$$
\begin{aligned}
\text{S.0}: \ & \text{st} = \text{rq} \wedge \\
& \text{seq.p} = \text{seq.q} \wedge \\
& \#\text{ch.p.q} + \#\text{ch.q.p} = 0
\end{aligned}
$$

When the protocol is in the state of S.0, the only action that is enabled for execution is process p's first action. Executing this action takes us to state S.1:

$$
\begin{aligned}
\text{S.1}: \ & \text{st} = \text{rp} \wedge \\
& \text{seq.p} = \text{seq.q} \wedge \\
& (\forall \text{i}: \ 0 \leq \text{i} < \text{n}: \ \text{c.p}[\text{i}+1] = \text{H}(\text{c.p}[\text{i}])) \wedge \\
& \text{rem} = \text{n}
\end{aligned}
$$

[2] Strictly speaking, all states mentioned are a set of infinite states.

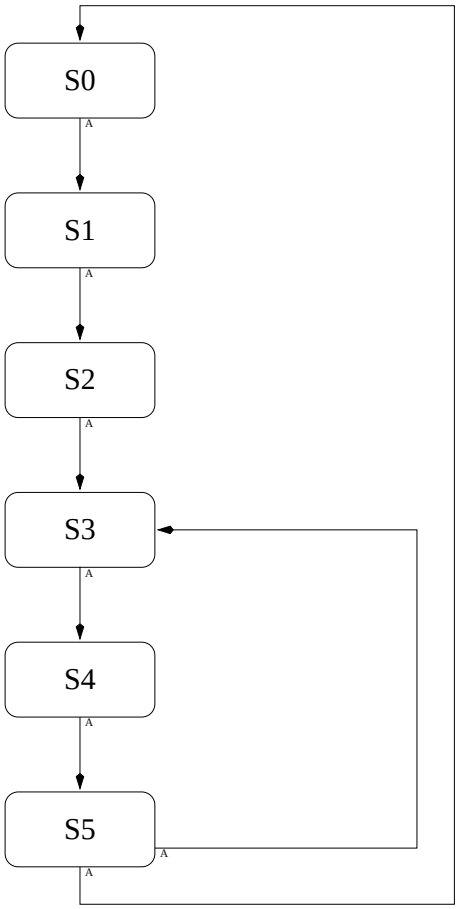

Figure 4: State-transition diagram of Basic Payword Protocol assuming no communication errors can take place and the messages exchanged between the two processes cannot be removed, modified or replayed by an adversary. Also note that the transitions from one state to another can only occur if an action is executed from the protocol. These action executions are denoted by "A".

ch.p.q = {rqst(NCR(sk, (c.p[n]; seq.p)))} ∧
#ch.q.p = 0

At any state that satisfies S.1, only the first action in q is enabled. This is because p is in *rp* state and cannot send either a request or a payment. Also, there is a message in the channel so p cannot timeout. Consequently, process q's execution of its first action brings the protocol into a state satisfying S.2:

S.2 : st = rp ∧
seq.p + 1 = seq.q ∧
(∀i: 0 ≤ i < n: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n: c.q[i + 1] = H(c.q[i])) ∧
akn.q = c.p[n] ∧
rem = n ∧
pos = 0 ∧
#ch.p.q = 0 ∧
ch.q.p = {rply(NCR(sk, (c.q[n]; seq.p)))}

Similarly, execution from state S.2 can only be followed by execution of the second action in p leading to state S.3. And S.3 enables the third action in p, modifing the state of the protocol to state S.4. These states are:

S.3 : st = py ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[rem] ∧
rem + pos = n ∧
#ch.p.q + #ch.q.p = 0

S.4 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[rem + 1] ∧
rem + pos = n - 1 ∧
ch.p.q = {pay(c.p[rem])} ∧
#ch.q.p = 0

Finally, the second action in q changes the state of the protocol from S.4 to S.5. The state S.5 is given below:

S.5 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[rem] ∧
rem + pos = n ∧
#ch.p.q = 0 ∧
ch.q.p = {ack(c.q[n - pos])}

From S.5, the protocol goes back to either S.0 or S.5.

Let S denote the state predicate: S.0 ∧ S.1 ∧ S.2 ∧ S.3 ∧ S.4 ∧ S.5. Then, S is defined as the correct execution of the Basic Payword Protocol and the states comprising S can be known as "good states."

The state S, satisfies the following three conditions:

1. *Closure under Execution*: If the state of the protocol satisfies S, then execution of any action of the protocol will also yield the protocol in a state that satisfies S.

2. *Reachability*: Since the initial state satisfies S and the protocol is closed under execution, it follows that every reachable state in the protocol satisfies the predicate S.

3. *Enablement*: At each state that satisfies S, atleast one action is enabled for execution in the protocol.

The overall state diagram can be seen in figure 4. This illustrates that the protocol is correct if an adversary does not disrupt the communication.

4.2 Security of the Basic Payword Protocol

Now that we proved that the protocol is correct in the absence of an adversary, we shall verify the security of the protocol in the presence of an adversary. An adversary can only perform one of three actions: message removal, message modification or message replay.

Overcoming Message Loss

An adversary can only remove messages when the protocol is in the states S.1, S.2, S.4, and S.5 because these are the only states that contain messages in the channels.

Let's look at state S.1 and S.2. When the protocol is in either state and a message is lost, p timeouts in both cases and resends a request message. However, if p resends a request message that process q already had received, then q simply resends the reply without computing the chain or setting its variables. Therefore the state S.1 needs to be modified to reflect that:

$$
\begin{aligned}
\text{S.1}: \; & \text{st} = \text{rp} \wedge \\
& (\text{seq.p} = \text{seq.q} \vee \\
& \qquad (\text{seq.p} + 1 = \text{seq.q} \wedge \\
& \qquad \text{akn.q} = \text{c.p[n]} \wedge \\
& \qquad (\forall \text{i}: \; 0 \leq \text{i} < \text{n}: \; \text{c.q[i} + 1] = \text{H(c.q[i])}) \wedge \\
& \qquad \text{pos} = 0)) \wedge \\
& (\forall \text{i}: \; 0 \leq \text{i} < \text{n}: \; \text{c.p[i} + 1] = \text{H(c.p[i])}) \wedge \\
& \text{rem} = \text{n} \\
& \text{ch.p.q} = \{\text{rqst(NCR(sk, (c.p[n]; seq.p)))}\} \wedge \\
& \#\text{ch.q.p} = 0
\end{aligned}
$$

Notice that the disjunction is added to S.1. If q's sequence number is greater than p's by one, q's chain has already been computed, and that q's *akn* and *pos* are set to appropriate values. Otherwise, the request message in the channel is a new one that q has not received previously.

The state L.0, indicates that a request message has been lost. The state L.0 is the same as S.1 except that the channels are empty:

$$
\begin{aligned}
\text{L.0}: \; & \text{st} = \text{rp} \wedge \\
& (\text{seq.p} = \text{seq.q} \vee \\
& \qquad (\text{seq.p} + 1 = \text{seq.q} \wedge \\
& \qquad \text{akn.q} = \text{c.p[n]} \wedge \\
& \qquad (\forall \text{i}: \; 0 \leq \text{i} < \text{n}: \; \text{c.q[i} + 1] = \text{H(c.q[i])}) \wedge \\
& \qquad \text{pos} = 0)) \wedge \\
& (\forall \text{i}: \; 0 \leq \text{i} < \text{n}: \; \text{c.p[i} + 1] = \text{H(c.p[i])}) \wedge \\
& \text{rem} = \text{n} \wedge \\
& \#\text{ch.p.q} + \#\text{ch.q.p} = 0
\end{aligned}
$$

If a reply message have been lost, it is nearly the same as L.0. The only difference is that the first disjunct, $seq.p = seq.q$, above is false. In other words, the case where the reply is lost, is a stronger predicate than L.0. Consequently, that state satisfies L.0 and thus can be combined into a single state, L.0.

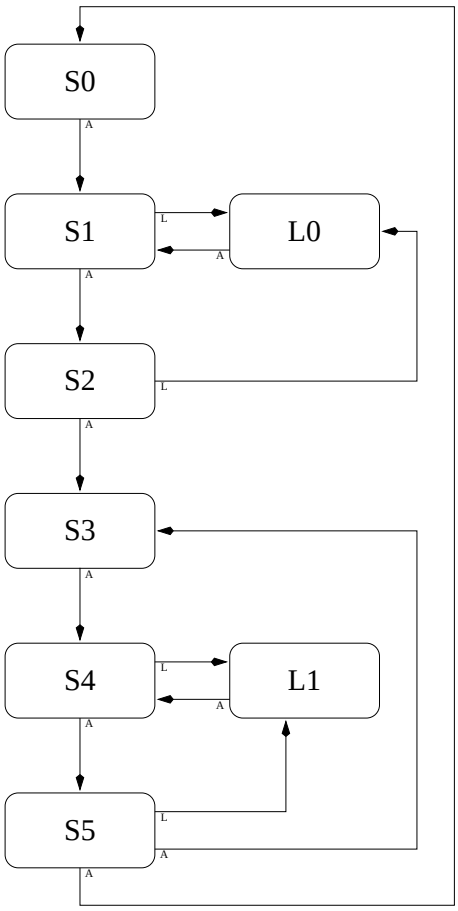

Figure 5: The state-transition diagram that overcomes message loss while preserving the correctness of the protocol. Observe that the label "L" is used to distingish transitions executed by an adversary from transitions through action execution.

Let us look at state S.4 now. If the protocol is in state S.4, it is possible that the payment p sent was received by q but the acknowledgment never made it through. If that is the case, we would have $rem + pos = n$. As before, the state S.4 should reflect this. The predicate S.4 needs to be modified to:

S.4 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[n - pos] ∧
(rem + pos = n - 1 ∨
rem + pos = n) ∧
ch.p.q = {pay(c.p[rem])} ∧
#ch.q.p = 0

The state L.1, that accounts for loss of a payment, is exactly the same as S.4 above except that the channels are empty:

L.1 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[n - pos] ∧
(rem + pos = n - 1 ∨
rem + pos = n) ∧
#ch.p.q + #ch.q.p = 0

Again, if the acknowledgment was lost, the first disjunct above would be false but the predicate would be satisfied. Therefore, we can merge the two states into one.

Hence, the protocol can transition from either state S.1 or S.2 to L.0. Similarly, transitions to L.1 can be made from S.4 or S.5. At either states L.0 or L.1, only the timeout action is able to execute in process p. Execution of this action from state L.0 yields the protocol in state S.1. If the state of the protocol were in L.1, the protocol moves on to state S.4.

Figure 5 shows the Basic Payword Protocol with the added transitions for message loss. It can be seen that the protocol is able to recover from message loss because each transition to either L.0 or L.1 takes the protocol back to a state satisfying S. This completes our proof for message loss.

Overcoming Message Modification

Now, we can assume that the adversary can arbitrarily modify messages. Message modification can occur if there is a request, reply, payment or acknowledgment in the channels. Let us look at each case.

The request message has the following property:

rqst(NCR(sk, (x; y))) where (y=seq.q ∨ y=seq.q - 1)

which limits the integer y to a very small subset of possible values. Therefore, if the message is arbitrarily modified, it would satisfy the negative predicate, given below:

rqst(NCR(sk, (x′; y′))) where (y′≠seq.q ∧ y′≠seq.q - 1)

Similarly, the reply, payment and acknowledgment satisfy the following predicates:

rply(NCR(sk, (x; y))) where (y=seq.p)
pay(x) where (x=c.p[rem])
ack(x) where (x=c.q[n - pos])

and again the above predicates are a small subset of states. Arbitrary modification negates these predicates to:

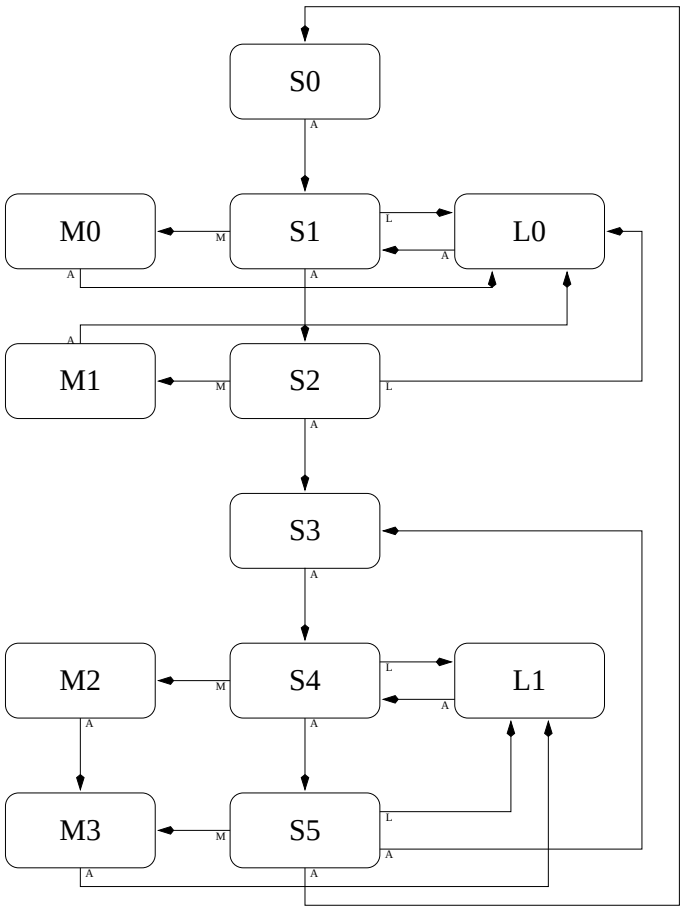

Figure 6: The states for modification are added here to show how the protocol handles random changes to messages between the channels. As can be seen above, the protocol is able to recover from modification or "M" transitions and go back to valid states.

rply(NCR(sk, (x′; y′))) where (y′≠seq.p)
pay(x′) where (x′≠c.p[rem])
ack(x′) where (x′≠c.q[n - pos])

Consequently, the four modification states are:

M.0 : st = rp ∧
(seq.p = seq.q ∨
(seq.p + 1 = seq.q ∧
akn.q = c.p[n] ∧
(∀i: 0 ≤ i < n: c.q[i + 1] = H(c.q[i])) ∧
pos = 0)) ∧
(∀i: 0 ≤ i < n: c.p[i + 1] = H(c.p[i])) ∧
rem = n ∧
ch.p.q = {rqst(NCR(sk, (x; y)))} where (y ≠ seq.q ∧ y ≠ seq.q - 1) ∧
#ch.q.p = 0

M.1 : st = rp ∧
seq.p + 1 = seq.q ∧
(∀i: 0 ≤ i < n: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n: c.q[i + 1] = H(c.q[i])) ∧
akn.q = c.p[rem] ∧
rem = n ∧
pos = 0 ∧
#ch.p.q = 0 ∧
ch.q.p = {rply(NCR(sk, (x; y)))} where (y ≠ seq.p)

M.2 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[n - pos] ∧
akn.q = c.p[n - pos] ∧
(rem + pos = n - 1 ∨
rem + pos = n) ∧
ch.p.q = {pay(x)} where (x ≠ c.p[rem]) ∧
#ch.q.p = 0

M.3 : st = ak ∧
seq.p = seq.q ∧
(∀i: 0 ≤ i < rem: c.p[i + 1] = H(c.p[i])) ∧
(∀i: 0 ≤ i < n - pos: c.q[i + 1] = H(c.q[i])) ∧
akn.p = c.q[rem] ∧
akn.q = c.p[rem] ∧
rem + pos = n ∧
#ch.p.q = 0 ∧
ch.q.p = {ack(x)} where (x ≠ c.q[n - pos])

As can be seen from inspection, each of the above states enables only one action of the protocol. The state M.0 will cause q to discard the message and M.1 will cause p to discard the message. State M.2 will cause q to send a bad acknowledgment and the process will go to state M.3. Upon reaching M.3, process p discards the message. In other words, each modification of a message is converted to message loss. This is illustrated in figure 6.

Since we already proved that the protocol is able to continue in case of message loss, this completes the proof that the protocol is tolerant to message modification.

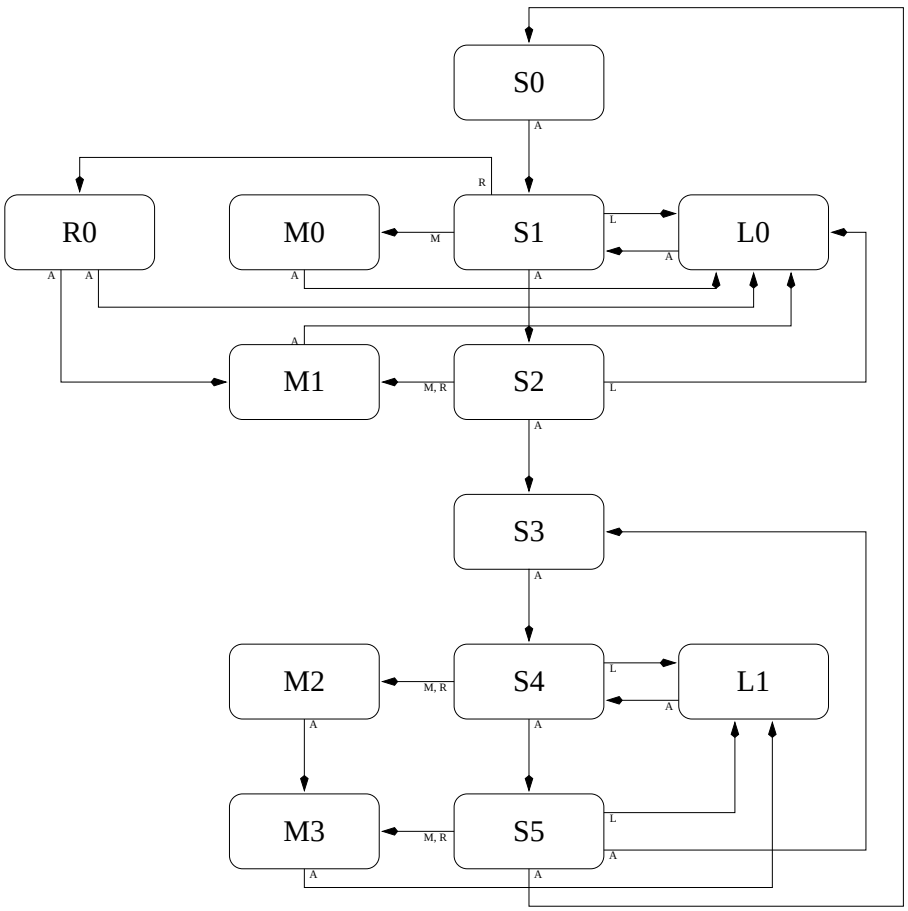

Figure 7: This is the complete state-transition diagram. It includes "A" transitions that are executed by actions in the protocol, "L" transitions denote message loss, "M" indicates modification of messages and "R" show replay of messages.

Overcoming Message Replay

The final step to prove that the Basic Payword Protocol is secure is to prove that the protocol is correct if an adversary replays any messages. As before, let us look at the predicates that satisfy each message in the channel:

rqst(NCR(sk, (x; y))) where (y=seq.q $\vee$ y=seq.q - 1)
rply(NCR(sk, (x; y))) where (y=seq.p)
pay(x) where (x=c.p[rem])
ack(x) where (x=c.q[n - pos])

When any of those messages are replaced with a prior message, it satisfies the following predicate:

rqst(NCR(sk, (x′; y′))) where (y<seq.q)
rply(NCR(sk, (x′; y′))) where (y<seq.p)
pay(x′) where (x′=c.p[y′] for y′<rem)
ack(x′) where (x′=c.q[y′] for y′<n - pos)

From the predicates, it can be seen that the second, third and fourth predicate are weaker than the corresponding second, third and fourth predicates for message modification. Therefore, we can argue that if an adversary replays the reply, payment or acknowledgment message, it will go to a state satisfying M.1, M.2 and M.3 respectively.

The predicates for replay of request and payment can be stated as follows:

$$
\begin{aligned}
&\text{R.0}: \text{st} = \text{rp} \wedge \\
&\quad (\text{seq.p} = \text{seq.q} \vee \\
&\qquad (\text{seq.p} + 1 = \text{seq.q} \wedge \\
&\qquad \text{akn.q} = \text{c.p[n]} \wedge \\
&\qquad (\forall \text{i}: 0 \leq \text{i} < \text{n}: \text{c.q[i+1]} = \text{H(c.q[i])}) \wedge \\
&\qquad \text{pos} = 0)) \wedge \\
&\quad (\forall \text{i}: 0 \leq \text{i} < \text{n}: \text{c.p[i+1]}=\text{H(c.p[i])}) \wedge \\
&\quad \text{rem} = \text{n} \wedge \\
&\quad \text{ch.p.q} = \{\text{rqst(NCR(sk, (x; y)))}\} \text{ where } (\text{y} < \text{seq.q}) \wedge \\
&\quad \#\text{ch.q.p} = 0
\end{aligned}
$$

Once more, each replay enables a single action in the protocol. If a request message is replayed and the sequence number is one less from the sequence number from q's, q generates a bad reply message. Otherwise, q discards the message.

Also, a bad payment causes q to send a bad acknowledgment. When process p gets this bad acknowledgment, it discards it. Later, it timeouts and resends the payment. On the same token, replaying an acknowledgment causes p to discard it. Process p will resend the payment and q will resend the acknowledgment.

Therefore, we proved that the protocol is able to recover from message loss, modification and replay. Considering that an adversary can only perform one of the three actions for a finite amount of time, the protocol will eventually move from the bad states to good states.

4.3 Protection of the Basic Payword Protocol

So far, we have proved that the protocol is able to go back to good states from bad states. To complete the proof, we must also show that the internal state of the protocol is not modified when the protocol deviates from the good states.

To do so, we must do an inspection of each guard that is executed when the protocol is in a bad state. If execution of any of the guards causes either process to change its critical[3] variables, then the protocol is incorrect.

Examination of the Basic Payword Protocol specification reveals that the critical variables for p and q are:

$$
\begin{aligned}
&\text{p}: st, c, seq, akn, rem \\
&\text{q}: c, seq, akn, pos
\end{aligned}
$$

In all cases, both processes either discard incorrect messages or respond with incorrect messages. They do not, however, modify the above variables. This completes the formal proof that the protocol is correct.

5 Variable Payword Protocol

5.1 Overview

The previous protocol allows little flexibility. In particular, if process p would like to send a large number of paywords, it would have to do so one at a time. To allow many paywords to be sent at one time, we could add another field for each payment that would represent the number of payments.

By adding this second field, we introduce the problem that we brought up in section 1.2. That is, an adversary can hash the payword a certain number of times and decrement the second field by the same amount.

[3] A variable that is not used for temporary storage. It usually stores a value that is extremely important for the overall program to function.

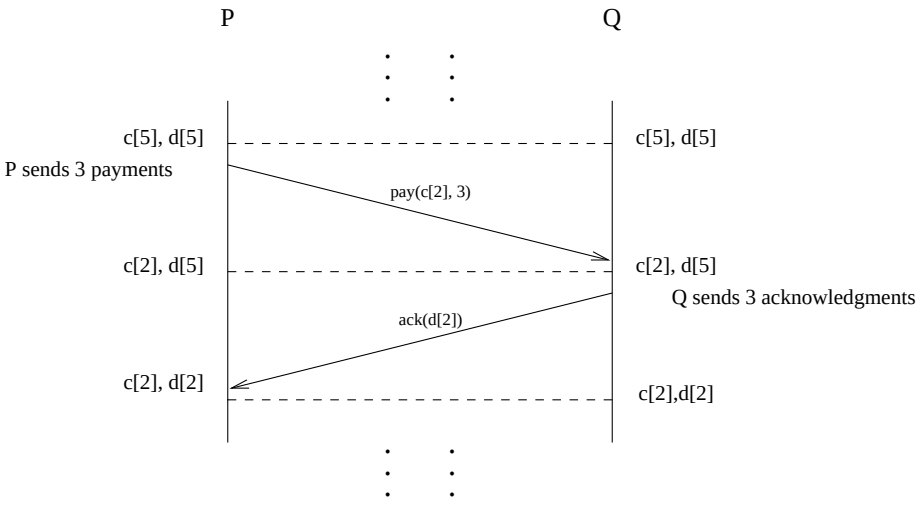

Figure 8: An example of an exchange of payments in the Variable payword protocol where each payment is followed by an acknowledgment of the same amount.

The above problem was not present in the Basic Payword Protocol because each payment was created by a single hash function applied to the previous payment. If an adversary hashed any payment, it would equal a previous payment and q would resend an acknowledgment without assuming that the payment is a new one.

5.2 Solution to the Error

Let's consider the protocol that we have developed so far:

$p \rightarrow q : c[n-x]; x$
$q \rightarrow p : c[n-1]$

Here, x can be set to any arbitrary value between 1 and the remaining length of the chain. When q receives the payment, it would apply the hash function x number of times on $c[n-x]$ to authenticate the payword. Process q would then send an acknowledgment.

While this solution is better than sending and receiving x number of payments and acknowledgments over the network, it still does not solve the problem. An adversary would still hash the paywords and reduce the total number of paywords exchanged. Again, neither processes would be able to detect this.

We must either communicate to q what p sent without risk of modification by or let p know what q received. Achieving the former would add considerable computational overhead. For example, encrypting each payment would be too inefficient and defeat the purpose of having a light-weight micro-payment scheme. Therefore, we try the latter.

One solution that we came up with is to let q communicate the number of payments received by modifying its acknowledgment as shown below:

$p \rightarrow q : c[n-x]; x$
$q \rightarrow p : c[n-x]$

The above exchange is pictured in figure 8.

When p gets an acknowledgment, it hashes the acknowledgment the same number of times as the prior payment. Process p then compares this result with akn. If the two are equal, p is assured that the payment it sent was received by q for the correct amount. Otherwise, p keeps resending the payment until it does receive an acknowledgment that satisfies the above condition. This is demonstrated in figure 9.

Although our solution does not guarantee that the payment received by q is of the correct amount, it does ensure that q will ultimately get the correct amount of payment.

We specified the variable payword protocol using the above solution.

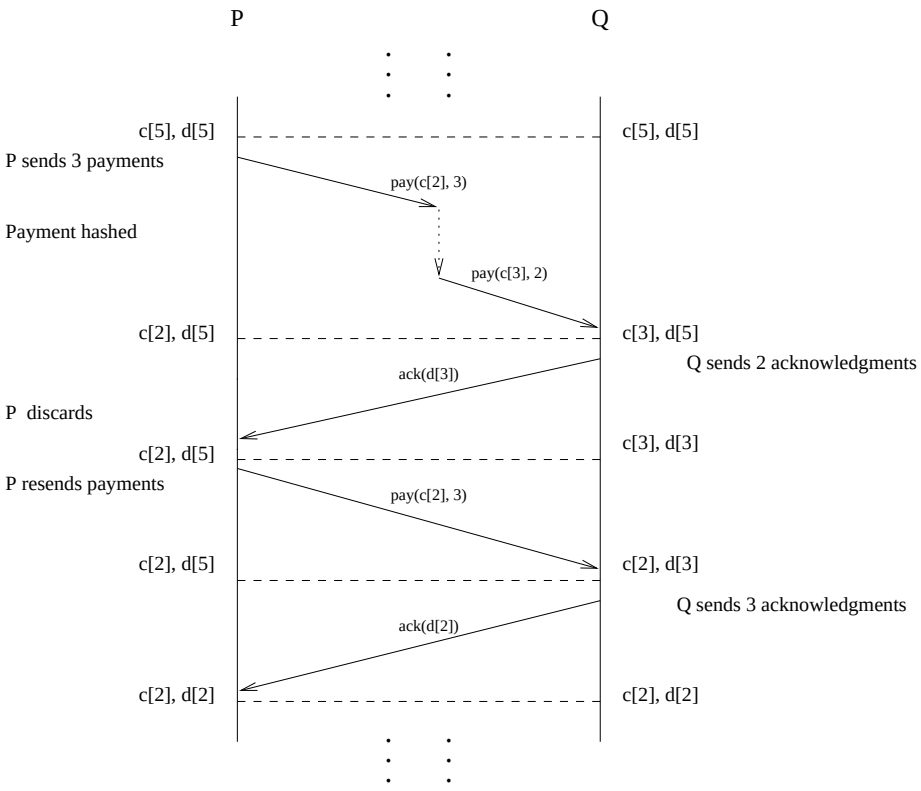

Figure 9: This shows how the Variable Payword protocol recovers if an adversary hashes payments and decrements the cumulative number of paywords sent by process p.

5.3 Abstract Protocol Specification

```
process p
    const sk, n
    var   st :              set {rq,rp,py,ak},              {init rq}
          c :               array [0..n] of integer,
          seq, akn, u, v :  integer                         {seq init 1}
          rem :             0..n,
          x :               1..n,
    begin
          st = rq ⟶ st, c[0], rem := rp, NONCE, 0;
                    do (rem<n) →
                              c[rem+1], rem := H(c[rem]), rem+1
                    od;
                    send rqst (NCR(sk, (c[n]; seq))) to q

    []    rcv rply(u) from q ⟶
                    u, v := DCR(sk, u);
                    if st=rp ∧ v=seq → st, seq, akn := py, seq+1, u
                    [] st≠rp ∨ v≠seq → skip
                    fi

    []    st = py ⟶ x := any;
                    x := min(x, rem);
                    st, rem := ak, rem-x;
                    send pay(c[rem], n-rem) to q

    []    rcv ack(u) from q ⟶
                    if akn=H^x(u) ∧ rem>0 → st, akn := py, u
                    [] akn=H^x(u) → st := rq
                    [] akn≠H^x(u) → skip
                    fi

    []    timeout (#ch.p.q + #ch.q.p = 0) ∧ (st=ak ∨ st=rp) ⟶
                    if st=rp → send rqst (NCR(sk, (c[n]; seq))) to q
                    [] st=ak → send pay(c[rem], n-rem) to q;
                    [] st≠rp ∧ st≠ak → skip
                    fi
    end.


process q
    const sk, n
    var   c :               array [0..n] of integer,
          seq, akn, t, u :  integer                         {seq init 1}
          pos :             0..n,
    begin
          rcv rqst(t) from p ⟶
                    t, u := DCR(sk, t);
                    if u=seq → seq, c[0], akn, t, pos := sq+1, NONCE, t, 0, 0;
                              do (t<n) →
                                   c[t+1], t := H(c[t]), t+1
                              od;
                              send rply(NCR(sk, (c[n]; u))) to p;
```

```
                [] u=seq-1 → send rply(NCR(sk, (c[n]; u))) to p;
                [] u≠seq ∧ u≠seq-1 → skip
                fi

[]      rcv pay(t, u) from p ⟶
                if akn=H^{u-pos}(t) → akn, pos := t, u;
                                      send ack(c[n-pos]) to p
                [] akn≠H^{u-pos}(t) → send ack(c[n-pos]) to p
                fi
end.
```

5.4 Discussion

Process p

Process p's specification is very similar to the previous protocol. In particular, the first two actions are identical. The notable differences are in the third and fourth action.

The third action is for sending payments. Here, x is set to an arbitrary number that is atleast 1 but no more than the length of the remaining chain. It then sends the payment and decreases the chain by the amount of the payment.

The acknowledgments received from q are handled differently, as well. Instead of hashing a single time, the hash value received is hashed x times and compared with the previous acknowledgment. If the two do not match, p stays in state *ak*, indicating that it is still waiting for an acknowledgment for the correct amount.

Process q

In process q, the first action is exactly the same as the Basic Payword protocol. Nevertheless, the second action is modified to accept another field in the payment. Also, instead of hashing a single time, q hashes $u - pos$ times. Here, u is the total number of paywords sent including the current payment and *pos* is the number of paywords received in the previous payment. The difference yields the number of paywords in the current payment.

6 Hierarchical Payword Protocol

6.1 Overview

The payword protocols presented earlier requires one hash operation for each payword. For example, if 100 paywords need to be sent, the total cost would be 100 hash operations for both p and q. This is considerably computation intensive. Therefore, we derived a variation, called hierarchical protocol.

The buyer, process p, uses two chains of paywords instead of one. The second chain of paywords equals, say 10 paywords of the first chain. Consequently, to send a 100 paywords, p would only send 10 paywords from the second chain, requiring only 10 hash operations for both processes.

6.2 Abstract Protocol Specification

```
process p
    const sk, n
    var     st :                set {rq,rp,py,ak},          {init rq}
            c, d :              array [0..n] of integer,
            seq, aknc, aknd :   integer                     {seq init 1}
            remc, remd :        0..n,
            x, y :              0..n,
            u, v, w :           integer
    begin
```

```
        st = rq ⟶ st, c[0], d[0], remc, remd := rp, NONCE, NONCE, 0, n;
                  do (remc<n) →
                          c[remc+1], d[remc], remc := H(c[remc]), H(d[remc]), remc+1
                  od;
                  send rqst (NCR(sk, (c[n]; d[n]; seq))) to q

    []  rcv rply(u) from q ⟶
                  u, v, w := DCR(sk, u);
                  if st=rp ∧ w=seq → st, seq, aknc, aknd := py, seq+1, u, v
                  [] st≠rp ∨ w≠seq → skip
                  fi

    []  st = py ⟶ x, y := any, any;
                  x, y := min(x, remc), min(y, remd);
                  st, remc, remd := ak, remc-x, remd-y;
                  send pay(c[remc], n-remc, d[remd], n-remd) to q

    []  rcv ack(u, v) from q ⟶
                  if aknc=H^x(u) ∧ aknd=H^y(v) ∧ (remc>0 ∨ remd>0) →
                                                  st, aknc, aknd := py, u, v
                  [] aknc=H^x(u) ∧ aknd=H^y(v) → st := rq
                  [] aknc≠H^x(u) ∨ aknd≠H^y(v) → skip
                  fi

    []  timeout (#ch.p.q+#ch.q.p = 0) ∧ (st=ak ∨ st=rp) ⟶
                  if st=rp → send rqst (NCR(sk, (c[n]; d[n]; seq))) to q
                  [] st=ak → send pay(c[remc], n-remc, d[remd], n-remd) to q;
                  [] st≠rp ∧ st≠ak → skip
                  fi
    end.


process q
    const sk, n
    var   c, d :              array [0..n] of integer,
          seq, aknc, ackd :   integer,                {seq init 1}
          posc, posd :        0..n,
          t, u, v, w :        integer
    begin
          rcv rqst(t) from p ⟶
              t, u, w := DCR(sk, t);
              if w=seq → seq, c[0], d[0], aknc, aknd, posc := sq+1, NONCE, NONCE, t, u, 0;
                         do (posc<n) →
                                 c[posc+1], d[posc], posc := H(c[posc]), H(d[posc]), posc+1
                         od;
                         posc, posd := 0, 0;
                         send rply(NCR(sk, (c[n]; d[n]; w))) to p;
              [] w=seq-1 → send rply(NCR(sk, (c[n]; d[n]; w))) to p;
              [] w≠seq ∧ w≠seq-1 → skip
              fi

    []    rcv pay(t, u, v, w) from p ⟶
              if aknc=H^(u−posc)(t) ∧ aknd=H^(w−posd)(v) → aknc, aknd, posc, posd := t, v, u, w;
                                                           send ack(c[n-posc], d[n-posd]) to p
```

[] aknc≠H^{u-posc}(t) ∨ aknd≠H^{w-posd}(v) → **send** ack(c[n - posc], d[n - posd]) **to** p
fi

end.

6.3 Discussion

This protocol is very similar to the variable payword protocol in section 5.3. Some notable differences in p are that p has to generate two chains before a request and sends two paywords with each payment.

The reply that process q sends also includes two roots so that q can acknowledge paywords from both chains. Hence, each acknowledgment from q contains two hash values.

Both processes need more variables for this protocol. Process p and q have *aknc* and *aknd* that store previous hash values of the other process. Also, process p uses *remc* and *remd* to keep track of the current position in the chain. Process q has *posc* and *posd* for the same purpose. Besides these changes, both processes have several additional variables to receive extra fields in the messages.

Observe that the hierarchical protocol can be modified to accomodate chains of more than 2. As a matter of fact, process p and q can utilize a hierarchy of an arbitrary number of chains.

7 Other Extensions of the Protocols

Note that the length of the chain in all the protocols is a fixed value of n. In some cases, it may be more suitable to allow the length of the chains to vary between, say, 1 to n where n is a constant. For example, if it is known beforehand that only k paywords will be exchanged between a certain buyer and seller, then it is more efficient to generate a chain of size k.

The payword protocols can easily be modified to create variable length chains. To do so, p would send the length of the chain based on estimated number of future transactions with q. Process p would pass this information to q so that q could create a chain of the same length for acknowledgments:

p → q : S⟨$c[n]; n; seq$⟩
q → p : S⟨$c[n]; seq$⟩

Another variation can be used to incorporate non-repudiation in the existing protocol. The request, that p sends to q, would include the vendor ID and then p would sign it with its private key and q's public key. Since, the shared key would not be employed, q could prove to a third party, namely a bank, that p promised to pay q a certain amount. This amount would be indicated by the product of the number of paywords sent by p and the value of each payword.

8 Conclusion and Status

We have presented the Basic Payword Protocol which allows efficient exchange of cash over a network. Other variations, like Variable Payword Protocol and Hierarchical Payword Protocol, allow exchange of cash of higher sums without a big penalty.

We found two errors in the original paper and developed the Basic Payword protocol that fixes the first error. The second protocol, Variable Payword protocol, offers a solution to the second problem. We also proved formally that the Basic Payword Protocol is correct.

Possible future work for these protocols includes attaining privacy, formal verfication of the Variable Payword Protocol and the Hierarchical Payword Protocol. We may also specify and verify other extensions of the payword protocol in the future.

References

1. Gouda, Mohamed G. *Security, Elements of Network Protocol Design*, Chapter 18, John Wiley & Sons, 1998.

2. Gouda, Mohamed G., Douglas H. Steves and Phoebe K. Weidmann. The Two-Nonce Protocol and its Formal Verification.

3. Herzberg, Amir and Hilik Yochai. Mini-Pay: Charging per Click on the Web.

4. Rivest, Ronald L., and Adi Shamir. PayWord and MicroMint: Two simple micropayment schemes. `http://theory.lcs.mit.edu/~rivest/RivestShamir-mpay.ps`