Evaluation of Micropayment Schemes

Ellis Chi

January 13, 1997 6:52 pm

Evaluation of Micropayment Schemes

Ellis Chi

This paper evaluates four micropayment schemes: Millicent, MicroMint, Payword, and Wenbo Mao's simple cash payment technique for the Internet. The paper first describes the differences between macro- and micropayment schemes, the notion of anonymity in electronic commerce, and common features used in payment protocols. It then examines details of the four payment protocols. The paper concludes with a summary of the features of the four payment schemes and compares their computation costs and storage requirements.

1.0 Introduction

This paper focuses on four micropayment schemes: Millicent [3], MicroMint [9], Payword [9], and Wenbo Mao's simple cash payment technique for the Internet [8]. It first provides some background about electronic payment schemes, including the differences between macro- and micropayment schemes, the trust model, the notion of anonymity, and common features used in most payment protocols. Sections 3.0-6.0 each describes one of the four payment protocols. The first part of each section introduces the payment protocol; it describes the data types of the protocol, a purchase, and a redemption. The second part evaluates the protocol, including forgery prevention, double spending detection, and other issues specific to each protocol.

This paper does not require readers to have prior knowledge of electronic payment schemes. Readers who design and implement micropayment schemes should also read the original paper of the protocol being implemented and the Internet working draft for Micro Payment Transfer Protocol (MPTP) [4], which shows some insights on design policy and data flow models for a payment scheme.

2.0 Background

This section provides some background by describing the differences between macro- and micropayment schemes. It also defines the entities involved in a payment, their relative trustworthiness, and the meaning of anonymity as it is used in this paper.

2.1 Difference Between Macro- and Micropayment Schemes

Of the two types of payments, macropayment schemes transfer larger sums of money for each transaction, so usually the security requirement is more rigorous. To date, public key cryptography is normally used in macropayment for authentication (to prevent forgery) and encryption (to preserve privacy of the data). Examples are Ecash of DigiCash [2] and iKP of IBM [6]. Besides public key cryptography, macropayment schemes use on-line broker activities to detect double spending prior to the acceptance of a payment by the

vendor. For example, Ecash requires both vendor's broker and customer's broker to be on-line to verify the transaction amount. NetBill [10] (though claimed to handle micropayments) requires the NetBill server to perform on-line verification and redemption.

The computational and storage costs of micropayment schemes are suitable for small payments (e.g., purchasing a web page). Compared with macropayment schemes, the computational times of micropayment schemes are less because they use one-way, collision-resistant hashing extensively and minimize the use of public key cryptography. As a rough estimate, hashing is roughly about 100 times faster than RSA signature verification, and about 10,000 times faster than RSA signature generation [11]. Besides hashing, micropayment schemes generally avoid on-line verification by the broker. This saves the broker on-line processing time and on-line storage requirements. In addition, micropayment protocols generally keep the customer-vendor relationship transient to avoid both the necessity of setting up account-based systems on vendor sites [3, 11] and a potential performance bottleneck for one-time or infrequent customer-vendor interactions.

The security of micropayment schemes is apparently not as good as that of macropayment schemes. However, if a micropayment scheme is designed so that a customer only loses a few cents when his transaction is tampered with, and the cost of counterfeiting a coin is either computation- or policy-wise higher than the value of the coin, then the security is considered to be adequate.

In short, the classification of both micro- and macropayment schemes is based on processing time and storage requirements [4]. While macropayment schemes are more concerned with the authenticity and privacy of data and therefore need demanding encryption algorithms and on-line processing, micropayment schemes aim at providing a decent level of security for transactions with more economical time and storage requirement.

2.2 Entities and Trust Model

Payment schemes usually have three entities: a customer or a user (*U*), a bank or a broker (*B*), and a vendor (*V*). Occasionally a certificate authority (*CA*) appears in a payment protocol, and the *CA* is normally responsible for issuing certificates for identification. If it appears, the *CA* is the most trusted entity in a transaction. A broker keeps track of customers' and vendors' accounts and may certify (or issue) coins to customers and/or vendors. If the CA does not exist, the bank is the most trusted entity. The vendor is the next most trusted, and the customer the least trusted.

2.3 Anonymity

2.3.1 Definition

In this paper, a payment is anonymous if it conforms to at least one of the following criteria:

- It preserves the privacy of a customer's identity from a vendor so that the vendor cannot associate the customer's identity with his purchases[1].
- It preserves the privacy of a customer's identity from a broker.[2]

Under this definition, a cash payment is anonymous. It satisfies both criteria because a customer's identity is not linked to a transaction at all. However, an electronic payment often associates a user's identity with the transaction (e.g., a payment that requires a customer's signature). Such a connection discourages the customer from double spending and sometimes prevents a third party from forging a payment.

Even though an electronic payment requires a user identity, there are ways to retain anonymity. One way is to use an intermediary between any two parties in the payment process. An intermediary is a trusted third party who is responsible for hiding the identity of a sender from a receiver, auditing all the interactions, and revealing the identity of an entity only under controlled circumstances. An example of a payment protocol using an intermediary is described in [7].

Another way is to blind a customer's identity in a payment. If a protocol requires a bank to issue or certify money to a customer, the protocol should specify that the bank use a blinding technique (such as blind signatures [1]) to certify the money. During a payment, a customer uses the challenge-and-response technique [14]. A challenge-and-response technique normally hides the customer identity but can disclose it if the customer double-spends. However, using challenge-and-response will not help conceal the customer's identity when a vendor requires a customer's identity to complete a transaction (e.g., the customer's name and address for delivery). Nevertheless, if the protocol still requires anonymity to the broker, the vendor should blind the customer's identity at redemption.

2.3.2 Anonymity in Micropayment Schemes

Among the four micropayment protocols discussed in this paper, only the first version of MicroMint (in Section 4.1.1.1) and Wenbo Mao's payment scheme (in Section 6.0) offer anonymity. The first version of MicroMint is like a cash-based system while Wenbo Mao's payment scheme uses blind signatures and the vendor's cooperation to preserve the user's privacy from a broker. Other protocols in this paper are not anonymous according to the definition in Section 2.3.1; they all expose the customer's identity to the vendor and the broker.

2.4 Common Features

The following are some common features assumed in each micropayment scheme to be described.

Money generation: There are two ways of generating money for micropayment schemes.

- *Money is created or certified by a broker:* A customer is assumed to buy micropayment's money in bulk from a broker through a macropayment protocol, and the broker debits the customer's account. To both the customer and the broker (or whoever creates or certifies the money), this is a debit-based approach because the customer has to pur-

1. A user identity is defined as any form of representation consistently used to identity the customer to another entity. Under this definition, a pseudonym [1] is also considered a user identity.
2. It is almost impossible for a broker to know what a customer has purchased based on the amount that the customer has authorized the broker to pay a vendor.

chase a specific form of money in advance; whoever certifies the money will benefit from the "float." A return policy is required for a customer to refund or renew unused, expired money.

- *Money is generated by a customer:* The money generated by a customer may not need direct certification by a broker. In this case, no bulk purchase or macropayment scheme is required. The payment scheme is credit-based to a customer, vendor, and broker because the customer's account will not be debited until a redemption takes place.

Double spending detection: Database lookup is used in all payment protocols (either macro- or micro) to detect double spending. Different protocols have different sets of data to serve this purpose. In each of the following micropayment schemes discussed, this set of data will be specified in the subsection titled "Double Spending Detection."

Redemption: A redemption process is assumed to be off-line if it is required in a micropayment protocol.

Notion of time: Time is not discussed in any of the four payment protocols. However, each payment scheme uses time (e.g., an expiration date) as a factor to define the validity of the electronic money. Each must provide a definition of time in order to avoid a race condition.

Pay-before or pay-after: The micropayment protocols described in this paper do not address the issue of when a delivery takes place, whether before or after the payment; however, whether a protocol is pay-before or pay-after should not affect the mechanism of the payment scheme.

3.0 Millicent

The Millicent protocol was initiated by Mark Manasse at DEC SRC in 1995 and predated the other proposals reviewed in this paper. Millicent uses *scrip* and is a debit-based protocol to a customer, vendor, and broker.

3.1 Details of Millicent

3.1.1 A Vendor Scrip

A vendor scrip consists of two parts: the scrip body and its certificate.[1] *Scrip_body* consists of two parts:

- *id_material* consists of the vendor's ID, the scrip's ID, and the customer's ID
- *cert_material* contains the value of the scrip, the expiration date, and other parameters

1. A bank scrip and a vendor scrip have the same scrip structure. For the ease of specifying the parameters, this subsection focuses on only a vendor scrip.

The certificate of *Scrip_body* is the result of hashing *Scrip_body* and the master scrip secret (*MSS)* which is known only to the vendor (more specifically, the creator of the scrip). *H* is a one-way and collision-resistant hash function.

The vendor scrip appears as [3]:

$$S_V = Scrip\_body, H(Scrip\_body, MSS)$$
$$where$$
$$Scrip\_body = id\_material, cert\_material \qquad (EQ\ 1)$$

3.1.2 Buying Vendor Scrip

A broker can get a lot of vendor scrip in two ways. The broker can buy the scrip in bulk from a vendor, or the broker can get a license for scrip production. In the latter case, the broker buys the parameters required to generate vendor scrip and produces the scrip on demand. Licensing is a better approach than bulk purchasing the scrip because the vendor does not need to invest in resources for scrip production, and sending only the parameters reduces the network transmission time [3].

Before a customer makes a purchase from a vendor, he buys vendor scrip with broker scrip. The customer is assumed to get the broker scrip through a macropayment. The protocol for the exchange is similar to that for a purchase done between the customer and the vendor, and the protocol for a purchase is discussed in the next subsection.

3.1.3 Making Purchases

To make a purchase,[1] the customer sends to a vendor a request (*request*), the vendor's scrip (S_V), and the authenticator. The authenticator is the result of hashing *request*, S_V, and *customer_shared_key (CSK).*[2] *CSK* is given by the broker and is also known to the customer and the vendor.

$$C \text{—>} V : S_V, request, H(S_V, request, CSK) \qquad (EQ\ 2)$$

The vendor receives the message, derives *CSK*, and hashes it with S_V and the request. If the result is same as the hashed value received, the message is considered valid. To prevent double spending, the vendor looks up the scrip in the vendor's database. If still valid, the vendor sends back a reply and, if any, the change S'_V. The message looks like this:

$$V \text{—>} C : S'_V, reply, H(S'_V, reply, cert, CSK). \qquad (EQ\ 3)$$

This message is also authenticated by *CSK*; *cert* is used to verify the correspondence of the reply to the request. To make the next purchase, the customer uses S'_V.

1. The proposal outlined in [3] describes a set of Millicent protocols; the one presented here is more suitable for a micropayment based on the classification in Section 2.1 on page 5.
2. In the original paper [3], *customer_shared_key* is called *customer_secret*. The original name implies that the parameter is known only by the customer. As a result, the word "shared" is added for sake of clarity; "key" is used instead of "secret" to show that the parameter is known by more than one entity.

3.1.4 Multiple Brokers

The Millicent protocol can be used in a multiple broker environment. Typically a customer is trying to make a purchase from a vendor who has an account with a different broker [3]. The following steps are shown in Figure 1:

1. The customer *C* asks his own broker B_C for vendor scrip S_V.
2. B_C finds out that *V* does not have an account with him, so he asks *V* if he would like to open one.
3. *V* has his own broker B_V, so he gives B_C a contact for B_V.
4. B_C buys B_V's scrip and sends it to *C*.
5. *C* then buys the vendor's scrip from the vendor's broker.

This model has some flexible features. First, the vendor does not need to open an account with every Millicent broker (step 2). Second, the customer's broker does not buy the vendor's scrip directly from the vendor (although the broker could have in step 2). This is because having the broker make purchases from each outside vendor increases the network traffic and possibly the storage requirement of the broker's server, and having the broker get the vendor's scrip directly requires a macropayment protocol if the vendor does not accept the customer's broker scrip (step 4). The same reason applies to having the vendor's broker give his own scrip to the customer's broker, rather than giving the scrip directly to the customer. This protocol is similar to a person buying foreign currency from his own bank before going to another country.

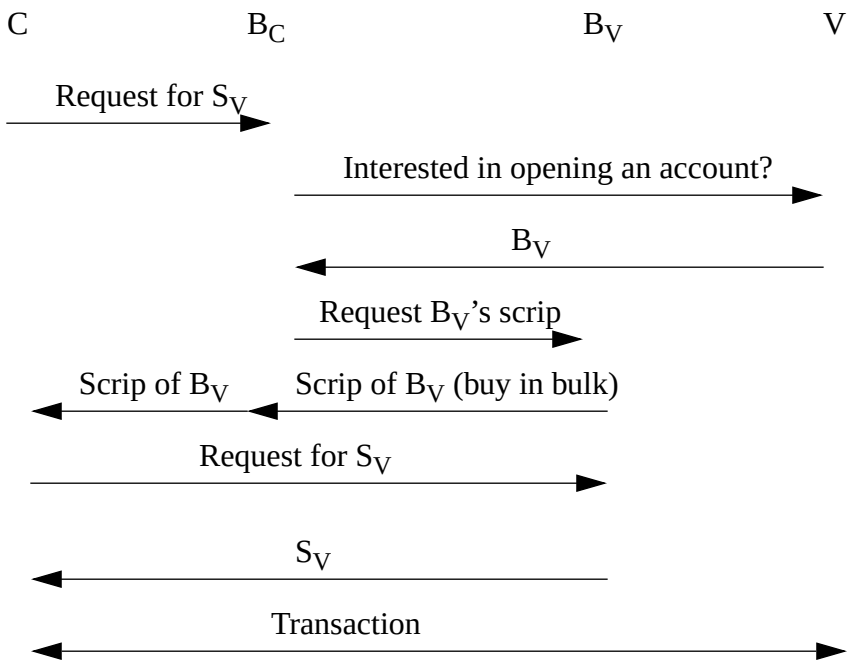

FIGURE 1. Buying vendor scrip from a different broker.

3.1.5 Redemption

The Millicent protocol does not require vendors to redeem for payment. This is because a customer always makes a purchase with a vendor's own scrip. The vendor gets the profit when he sells the scrip.

3.2 Evaluation

3.2.1 Forgery Prevention

A scrip hashes the master scrip secret, *MSS*, (known only to the owner of the scrip) with the scrip body to prevent the forgery of the scrip. A sender has to certify all messages by hashing them with a shared key (known only to the sender and the receiver) to prevent the forgery of the message.

3.2.2 Double Spending Detection

Double spending is found by looking up for duplicates of a valid scrip. The scrip's ID and *MSS* should be sufficient for double spending.

3.2.3 Other Comments

Some nice features of the Millicent protocol:

- *It is simple.* The use of scrip is easy to understand.
- *No public key encryption is used.* One hash is used for generating a valid scrip, one hash for sending the scrip over an insecure link, and one hash and one database lookup to verify the scrip.
- *Verification is decentralized.* While most of the payment schemes require vendors to redeem for payment with a bank off-line, Millicent vendors receive and verify their own scrip and do not need a broker to perform another verification.

Some undesirable features of the Millicent protocol:

- *It uses shared keys.* Millicent requires both a vendor and a broker to know the *customer_shared_key* (described in Section 3.1.3). It is okay for the broker to know about the key; however, having the vendor know about *CSK* requires the vendor to either maintain an extra database or perform an on-line query to the broker.
- *Scrip buyers can be spoofed.* Since only the owner of scrip knows about its secret, scrip buyers, including customers, cannot verify scrip.
- *A long-term relationship is assumed:* The Millicent protocol can be inconvenient if a customer tends to make infrequent purchases with vendors. He needs to go back to his broker and exchange for different vendor scrip for each transaction with a new vendor.

4.0 MicroMint

MicroMint was proposed by Ronald Rivest of MIT LCS and Adi Shamir of Weizmann Institute of Science. MicroMint is debit-based to a customer and a broker, while it is credit-based to a vendor. MicroMint uses difficult-to-produce *coins* “minted” by a broker.

4.1 Details of MicroMint

4.1.1 Coins

This subsection shows all three coin structures proposed in the original paper, even though the author of this paper intends to evaluate only the third one. The first two versions are simpler, and hopefully they help the third version seem more comprehensible to readers.

4.1.1.1 First Version: Generic Coins

Each coin is a *m*-bit *k*-tuple, $\{x_1,... x_k\}$, where each element (x_i) has the same *n*-bit hash value (*y*). This feature is called a *k-way collision*.

$$\begin{aligned} &Coin = \{x_1, ... x_k\} \\ &where \\ &h(x_i) = y \text{ for } i = 1, ...,k \end{aligned} \qquad \text{(EQ 4)}$$

h is a one-way, collision-resistant function. *m* is the length of an *x*-value, and *m* should be so large that storing all *x*-values possibly generated is not quite feasible [11]. As shown later, this is an important feature to prevent forgery. The broker avoids getting into the same problem during the production by secretly assigning a criterion for each coin circulated in a certain period of time, such as a month. This monthly criterion is presented as a fixed bit string ($\{b_n, ... , b_{n-t+1}\}$) in the first *t* bits of the hash values. Therefore, a valid coin now looks as follows:

$$\begin{aligned} &h(x_i) = y= \{b_n, ... , b_{n-t+1}, b_{n-t}, ... b_1\} \text{ for } i = 1, ..., k \\ &where \\ &\{b_n, ..., b_{n-t+1}\} = criterion\ of\ a\ month \end{aligned} \qquad \text{(EQ 5)}$$

This version of coins is a cash-based protocol and is anonymous (according to the definition in Section 2.3.1 on page 6) because the coins do not embed any customer’s information.

Forgery is discouraged in this version of coins; however, coins can easily be stolen and replayed because they are like cash that does not embed any user identification.

4.1.1.2 Second Version: User-Specific Coins

This version of coins contains user-specific information so that the coins are useful to only a particular customer.

In a user-specific coin, the *k*-tuple collides according to the following rule:

$$
\begin{aligned}
&Coin = \{x_1, \ldots x_k\} \\
&where \\
&h(x_i) = y_i \text{ ; for } i = 1, \ldots, k \\
&and \\
&y_{j+1} - y_j = d_j \bmod 2^u \text{; for } j = 1, \ldots, k\text{-}1 \text{ and } u = n - t \\
&and \\
&h(U) = (d_1, d_2, \ldots, d_{k-1})
\end{aligned}
\qquad \text{(EQ 6)}
$$

where U is the unique customer identity. Since the monthly criterion is enforced, all y_i vary only in the last u bits. This is reflected in the module difference in the above equation.

4.1.1.3 Third Version: User-Vendor-Specific Coins

Coins can be made user-vendor-specific to further discourage a customer or a malicious hacker from spending the same coin with other vendors.

As in the first and second versions, the hash value (y) in this version has n bits, and the first t bits of y gives the monthly criterion assigned by the broker. The remaining u bits can be divided into two: a (u-v)-bit upper part, y', and a v-bit lower part, y''. V is the unique vendor ID, and y''_1 is the unique coin ID. The vendor can redeem a coin only if it satisfies the monthly criterion (in EQ 7), the user's and the vendor's specification (in EQ 8 and EQ 9), and the uniqueness of the coin ID (y''_1) (in EQ 10). Figure 2 graphically shows the structure of a user-vendor-specific coin.

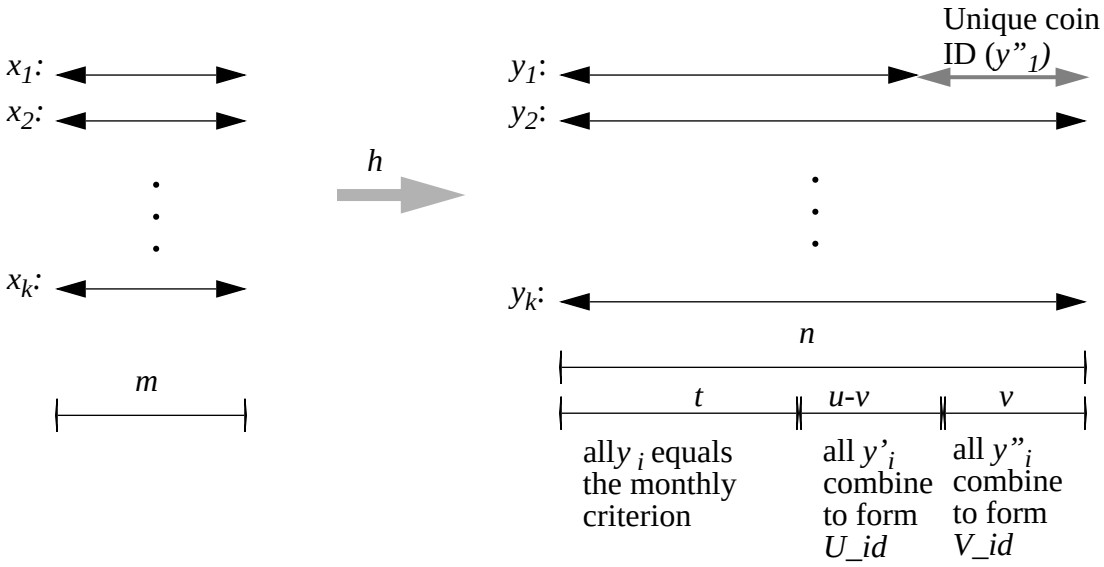

FIGURE 2. The structure of a coin. m is supposed to be larger than n. The scale is skewed to show the y-value details.

$$
\begin{aligned}
&Coin = \{x_1, \ldots x_k\} \\
&where \\
&h(x_i) = y_i = \{b_n, \ldots, b_{n-t+1}, b_{n-t}, \ldots b_1\}_i \text{ ; for } i = 1, \ldots, k \\
&and \\
&\{b_n, \ldots, b_{n-t+1}\}_i = criterion\ of\ the\ month
\end{aligned}
\qquad \text{(EQ 7)}
$$

and

$$y'_{i+1} - y'_i = d'_i \bmod 2^{u-v};\ \text{for } i = 1, ..., k-1$$
where
$$h'(U) = (d'_1, ..., d'_{k-1}) \qquad \textit{(EQ 8)}$$

and

$$y''_{i+1} - y''_i = d''_i \bmod 2^{v};\ \text{for } i = 1,..., k-1$$
where
$$h''(V) = (d''_1, ..., d''_{k-1}) \qquad \textit{(EQ 9)}$$

and

$$y''_1 \text{ is unique} \qquad \textit{(EQ 10)}$$

4.1.2 Buying User-Vendor-Specific Coins

A broker can organize all valid *x*-values into groups. In each group, the *x*-values have the same first $t+u-v$ bits. When a customer buys coins from the broker, he sells the groups that satisfy EQ 8.

4.1.3 Making Purchases

When the customer makes a purchase from a vendor, he picks one *x*-value out of each group of *x*-values and assembles a coin according to EQ 9 and EQ 10. The vendor checks the validity of the coin by hashing and prevents double spending by checking the uniqueness of y''_1.

The MicroMint paper [11] does not mention how coins are sent or received. It is believed that the payment is done either in cleartext or in a manner similar to the Millicent protocol shown in Section 3.1.3.

4.1.4 Redemption

The broker performs the same type of operations as the vendor to verify the coins. Since the vendor does not get the payment until the bank accepts the coins, MicroMint is a credit-based protocol from the vendor's viewpoint.

4.2 Evaluation

4.2.1 Forgery Prevention

A MicroMint broker prevents counterfeiting by making it a rarely profitable business. There are a few ways of forging coins. In one case, the forger tries to generate a coin for future months. Recall that an *x*-value has *m* bits, and its hash value has *n* bits. If the forger does not know the monthly criterion but wants to have the same capability of coin distribution as the broker, the forger has to store 2^m *x*-values in advance, while the broker needs to store only 2^{m-n+u}. If $m >> u$ and *m* is not a lot larger than *n*, the forger needs a lot more storage than the broker.

In another case, the criminal generates coins once the monthly criterion has been announced. To prevent this approach, the length of x-values can be made so long that it will take at least a month to generate a significant number of coins to make a profit.

However, this analysis is based on many assumptions. For example, it assumes that the forger is unable to save the unused x-values for the future.

4.2.2 Double Spending Detection

$y"_1$ in each coin serves as the unique identity of the coin and is used to detect double spending.

4.2.3 Other Comments

Some nice features of the MicroMint protocol are as follows:

- *No public key encryption is used.* Like Millicent, MicroMint does not use public key encryption. For the details of computation costs, see Table 2 on page 26.
- *No shared secret is required.* All information in a coin is public; therefore, it does not have the bottleneck problem that occurs in Millicent where the vendors and the brokers have to know customer shared secrets for verification.
- *The broker is off-line when a customer establishes a transaction with a new vendor.* Unlike Millicent, a MicroMint customer does not need to buy new vendor's scrip from a broker. This is also true for Payword and Wenbo's payment scheme discussed in the next two sections.
- *Users can verify coins.* Unlike Millicent, a MicroMint customer can verify his coins.

Some undesirable features of the MicroMint protocol are as follows:

- *Resources are wasted.* Generating coins well in advance rather than on demand can be a waste of resources.
- *Forgery prevention is based on many assumptions.* This problem was discussed in Section 4.2.1.

5.0 Payword

The Payword protocol is proposed together with MicroMint. Payword is a credit-based protocol to a customer, vendor, and broker. It is based on a chain of hash values, called *paywords* [11]. Each payword represents a particular denomination or unit of value.

5.1 Details of Payword

5.1.1 Paywords, Commitment, and User Certificate

Paywords are generated by a customer. They can be generated in advance or at the time of a purchase. To make a payword chain, a customer picks a random number as the nth pay-

word, w_n, and it is the "seed" for generating the rest of the paywords in the chain according to the following rule:

$$w_{i-1} = h(w_i) \qquad where\ i = 1, ..., n \qquad (EQ\ 11)$$

where h is a cryptographically strong function, such as MD5 [12], which has to be one-way and collision-resistant. The last value computed, w_0, is not part of the payword chain; it is the "root" of the chain and is embedded in a user-vendor-specific commitment.

A commitment authenticates w_0, and w_0 verifies the payword chain ($\{w_1,... w_n\}$). The chain is committed to a particular user-vendor relationship once its w_0 has been bound to a commitment (M). M is a signed message by a customer (shown in EQ 12). It consists of w_0, a vendor identity (V), the customer's certificate (C_U), an expiration date (D), and other information (I_M) necessary for the commitment.

$$M = \{w_0, V, C_u, D, I_M\}_{SK_U} \qquad (EQ\ 12)$$

Before any transaction takes place, the customer must get a user certificate from a broker. The certificate (C_U) authenticates the customer's public key, which is used to sign a commitment during a purchase. C_U is a signed message consisting of the broker's identity, the customer's public key, the expiration date, and other related information.

$$C_U = \{Broker\_id, customer\_public\_key, expiration\_date, other\_info\}_{SK_B} \qquad (EQ\ 13)$$

The three types of data work together to provide a secure purchase. A payword is sent unencrypted, but it is authenticated by a user-vendor-specific commitment. The user's public key used for signing the commitment is in turn authenticated by the user's certificate.

5.1.2 Making Purchases

First, a customer sends a vendor a commitment (M, defined in EQ 12). The vendor decrypts M with the customer's public key and verifies V and D. The customer signature is proven by C_U. Thus, a third party cannot forge a commitment by signing it with an invalid key, nor can he pretend to be the customer without knowing the customer's private key. If M is verified, the vendor stores it until it expires.

When the customer wants to make a purchase with one payword, he sends a pair, $P = (w_i, i)$, where $1 \le i \le n$. The vendor will see if $h(w_i)$ equals the payword previously sent. If verified, the vendor stores this last received payword pair (P_{last}), and the payment is considered valid. Figure 3 gives a summary of the Payword protocol (including redemption, which will be discussed in next subsection).

If a purchased item costs more than the value of one payword, the customer can pay more by skipping paywords. Assuming that the next unspent payword is w_{i+1}, each payword is worth one cent, and the item costs 5 cents, the customer can skip the first four unspent

paywords and send (w_{i+5}, $i+5$). The vendor verifies this pair by hashing the payword 5 times.

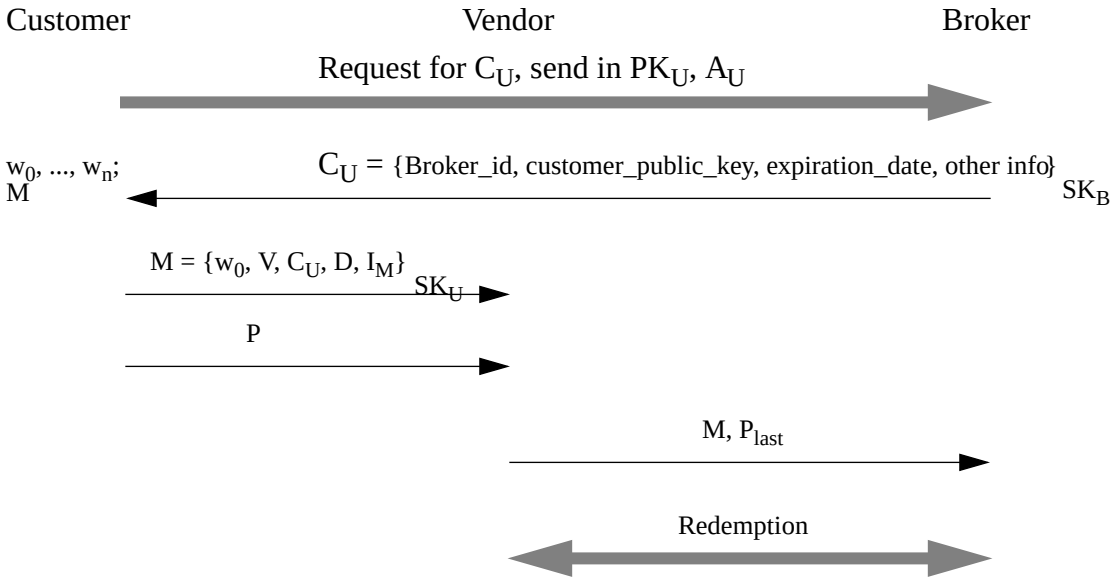

FIGURE 3. Summary of the Payword protocol. Thick arrows represent interactions using non-Payword protocols.

5.1.3 Redemption

A vendor only needs M and the last payword pair received *(P_{last})* for redemption. A broker verifies *M* and makes sure the last payword can be hashed into w_0 after *last* times. If everything looks right, the broker debits the customer's account and credits the vendor's.

5.2 Evaluation

5.2.1 Forgery Prevention

Spent paywords are the one-way hash values of the unspent paywords of the same chain; therefore, knowing the spent paywords should not expose the value of the unspent ones.

A payword chain is authenticated by a commitment, and it is signed by the customer. The identity of the customer is ensured by the user certificate (C_U), and it is signed by the broker.

5.2.2 Double Spending Detection

A payment in the Payword protocol consists of a commitment and its corresponding set of paywords. Therefore, double spending in Payword means replaying the same paywords with the same commitment. The last spent payword and the root allow a vendor or a broker to keep track of all the spent paywords of a commitment. Replaying valid commitments can be found by searching for duplicates in a database. A vendor should store all the valid commitments received to prevent a customer from replaying a valid commitment. The broker should do the same to prevent the vendor from double depositing.

The original paper [11] does not mention reusing paywords in different commitments; however, a customer should not do so, particularly, to the same vendor (described in detail in the next subsection).

5.2.3 Policy Model

The Payword protocol allows customers to generate their own money without requiring the bank to certify the paywords; this feature may cause a problem. Suppose customer A and customer B generated two sets of chains, and part of the chains contain the same paywords. If a vendor finds A and B turning in the same payword, and A happens to have spent the next few, the vendor now knows the next few unspent paywords in B's chain and can redeem more payment from the bank than he should have.

Therefore, Payword should be accompanied with a policy model to ensure that the paywords of a chain are distinct, and that every chain (and its w_0) ever generated is disjoint and nonadjacent to one another in the universe of the hash values. In other words, the policy model must carefully assign space to each customer for payword generation to prevent any overlaps. One possible solution is to incorporate user- and time-specific criteria in paywords; another one is to have users define their own hashing functions and send them along with payments.

5.2.4 Other Comments

The idea of paywords is adopted by the payment protocol proposed by Wenbo Mao (discussed in the next section) and the working draft of MPTP [4].

Some nice features of the Payword protocol are as follows:

- *Customers do not need to pay in advance.* Unlike Millicent, Payword is a credit-based system [11]. Instead of paying in advance, a Payword customer generates his own payword chain, and his account is debited only when the vendor redeems the payment.
- *Customers generate their own paywords.* This feature provides many flexibilities. For example, a customer can generate paywords based on demand. Also the customer can generate the exact amount for a payment.
- *It uses the latest state for verification.* Vendors and brokers only need to store all the valid commitments and the corresponding last-spent payword pairs. They can use hashing to derive all the paywords ever paid.

Some undesirable features of the Payword protocol are as follows:

- *Public key cryptography is required.* Signature generation and verification are required. See Table 2 on page 26 for details.
- *A long-term relationship is expected.* Payword is based on customer-vendor-specific payword chains that are suitable for long-term and frequent transactions between a customer and a vendor. However, Payword becomes inefficient if the relationship is one-time, e.g., random access of a rarely visited web site. In this case, one commitment is useful for only one payment, rather than being used for many payments, which can amortize the costs of public key cryptography.

- *Chains should be disjoint.* Paywords in each chain should be unique to prevent such problems as the one described in Section 5.2.3 from happening.

6.0 Wenbo Mao's Simple Payment Scheme

Wenbo Mao at Hewlett-Packard Laboratories, Bristol, proposed "Simple Cash Payment Technique for the Internet" (hereafter, referred to as the Wenbo Scheme). The Wenbo Scheme is an anonymous and non-vendor-specific Payword-based protocol. However, unlike Payword, it is debit-based to a customer and broker.

6.1 Details of the Wenbo Scheme

6.1.1 Coins, Bank Signature, Change, Spending Signatures, User Certificate

Coins in the Wenbo Scheme are the same as paywords in the Payword scheme. They are both generated by customers. In the Wenbo Scheme, the seed for generating a chain of coins and the root of the chain are notated as C_n and C_0, respectively.

In Payword, a commitment (a message signed by a customer) certifies a chain of coins. In the Wenbo Scheme, a commitment is carried out by a bank signature and a change of payment (which includes a signature from a vendor). These signatures allow a customer to use a chain of coins with any vendor. If a chain of coins is only used with one vendor, only a bank signature is ever needed for certifying the chain. This is the same as the Payword protocol. However, to allow a customer to pay the unspent coins with another vendor in the next payment, the previous vendor has to certify the unspent portion of the chain. Another way to think of the protocol is that a vendor has to sign the last coin being paid to him so that the next vendor knows how much of the chain has been consumed. A bank is just like another vendor to which the customer has paid a C_0.

This subsection will only talk about the structure of a bank signature; receiving change from a payment will be discussed later.

A bank signature (shown in EQ 14) certifies a chain of coins by signing the root (C_0) and the length of the chain (n). In addition, the bank certifies a user's identity (v), which is blinded by a one-way hash function (g) so that the real customer identity will not be revealed to the broker at redemption. Moreover, the message contains the hash value of the key used to generate the spending signature for the first payment ($g(x_1)$). A bank uses the blind signature technique [1] to sign the message, so the bank cannot relate the chain of coins to a customer's identity. When the message is signed, the bank also deducts the amount from the customer's account. Thus, the Wenbo Scheme is debit-based to customers and brokers.

$$\textit{Bank Signature}: \{C_0, g(v), g(x_1), n\}_{SK_B} \qquad \textit{(EQ 14)}$$

A spending signature is generated by a customer to authenticate himself in a payment and to certify how much he is paying to a vendor. It is used to detect double spending during a redemption (discussed in Section 6.2.2 on page 22). A spending signature is a customer-

vendor-specific signed message, and it is represented as a pair of values (*e, y*). This pair is generated using a variance of Schnorr's signature scheme. The use of Schnorr's signature scheme in the Wenbo Scheme can reveal a customer's identity if he double spends. For the details of how Schnorr's signature scheme works, refer to **Appendix B — Schnorr's Signature Scheme** on page 27. The spending signature for the first coin (C_1) is as follows:

$$\begin{aligned} &Spending\_Signature = (e, y) \\ &where \\ &e = H(m, g(x_1)) \text{ for } m = C_1, vendor\_id, timestamp \\ &y = f(e, other\ parameters) \end{aligned} \qquad \textit{(EQ 15)}$$

H is a one-way, collision-resistant hash function, and *f* is a signature-generating function of Schnorr's signature scheme.

A user certificate in the Wenbo Scheme, unlike Payword, is used as a parameter for verifying a spending signature and is sent in plaintext.

In a secure purchase, a coin is sent unencrypted, but a bank signature authenticates and certifies the initial chain of the coin. The customer uses a spending signature and a user certificate to vouch his payment to the vendor and to prove that the coin indeed belongs to him. Finally, a change certifies the unspent portion of the chain after a payment.

6.1.2 Making Purchases

6.1.2.1 Spending Coins with One Vendor

Suppose a customer generates a one-coin chain and pays the coin to a vendor. In this situation, the customer will not receive a change from the vendor. The customer sends to the vendor the bank signature, the spending signature, the customer's certificate issued by a certificate authority, the coin, the number of coin(s) spent (in this case, 1), and a timestamp.

$$\begin{aligned} &Bank\_Signature, \\ &C \text{ —> } V_1 : Spending\_Signature, Cert(v)^{1} \\ &C_1, 1, timestamp \end{aligned} \qquad \textit{(EQ 16)}$$

The vendor needs to verify three things. First, he uses Schnorr's signature scheme to verify if the cleartext portion of the message was originated by the customer (by using the spending signature and the customer's public key (*v*)). Second, the vendor makes sure that C_1 is certified by the bank (by trying to hash C_1 into C_0 found in the bank signature). Third, the vendor ensures that C_1 belongs to the right customer by matching the user identity in the bank signature with that in the spending signature.

If the customer is paying more than one coin, the treatment is the same as in Payword. The customer simply replaces the first coin and the index with other values.

1. In Wenbo's paper [8], the certificate is *Cert(customer_id, v)*. It was found that the *customer_id* is not necessary in the protocol as the result of a discussion between Wenbo Mao and the author [9].

6.1.2.2 Making a Purchase with the *k*th Vendor and Getting Change for the Payment

Suppose the customer has spent C_j coins and is spending i coins with the kth vendor, the customer sends the following to the kth vendor:

$$\begin{aligned} & \textit{Bank\_Signature}, \{C_j, g(x_k), n-j\}_{SK_{V_{k-1}}}, Cert(V_{k-1}), \\ C \text{ —> } V_k : \; & \textit{kth\_Spending\_Signature, Cert(v)}, \\ & C_{j+i}, \textit{ i, timestamp}, \\ & g(x_{k+1}) \end{aligned} \qquad \textit{(EQ 17)}$$

As in EQ 16, the first line consists of the bank's signature for the chain. The bank signature is required in each payment to show that the chain was once certified by a bank. The additional information, $\{C_j, g(x_k), n-j\}_{SK_{V_{k-1}}}$ and $Cert(V_{k-1})$, is the change from the previous payment. The signature of the change certifies that the chain has been used up to the jth coin. The second and third lines are similar to those in EQ 16. The fourth line is added to provide V_k the key for the next payment $g(x_{k+1})$. It will be embedded in the change (certified by V_k) for the current payment.

If the payment is valid and the chain is not all spent, the vendor needs to send the change for the payment to the customer. The change consists of the vendor's certificate and a signature. The signature is the same as a bank signature except that this signature does not contain the user's identity:

$$V_k \text{ —> } C : \textit{Change} = \{C_{j+i}, g(x_{k+1}), n-j-i\}_{SK_{V_k}}, Cert(V_k) \qquad \textit{(EQ 18)}$$

6.1.3 Redemption

To redeem for a payment, V_k sends the following to the broker:

$$\begin{aligned} & \textit{Bank\_Signature}, \{C_j, g(x_k), n-j\}_{SK_{V_{k-1}}}, Cert(V_{k-1}), \\ V_k \text{ —> } B : \; & \textit{kth\_Spending\_Signature, timestamp}, \\ & \textit{Change}, \\ & E(z) \end{aligned} \qquad \textit{(EQ 19)}$$

The first two lines come from the payment of the customer to V_k. *Bank_Signature* provides the certification from a bank. $\{C_j, g(x_k), n-j\}_{SK_{V_{k-1}}}$ gives the starting index for redemption. The *kth_Spending_Signature* is used when double spending occurs. *timestamp* is believed to indicate the "freshness" of the payment. *Change* shows the end index for redemption. *E(z)* embeds the customer's identity, which can only be decrypted by a certificate authority. The certificate authority discloses the customer's identity only if the broker proves that the customer colludes with a vendor.

The broker first verifies the bank signature. Then he hashes C_{j+i} i times. If it is equal to C_j in the *k-1*th vendor's signature, the broker hashes it j more times to see if it equals C_0. Then the broker checks if the range of coins $\{C_{j+1}, \ldots C_{j+i}\}$ has already been redeemed. This happens if the customer has double spent or the vendor double deposited. These two situations can be detected and are discussed in the next subsection. If the redemption is valid, the broker credits the vendor's account and stores all the signed information for

detecting possible future frauds. Note that the broker cannot verify the spending signature because he does not know the customer's identity in the form required for Schnorr's signature scheme verification. The bank signature gives *g(v)*, while the signature verification requires *v*. Thus, the broker can only check if the spending signature is duplicated. A summary of the Wenbo Scheme is shown in Figure 4.

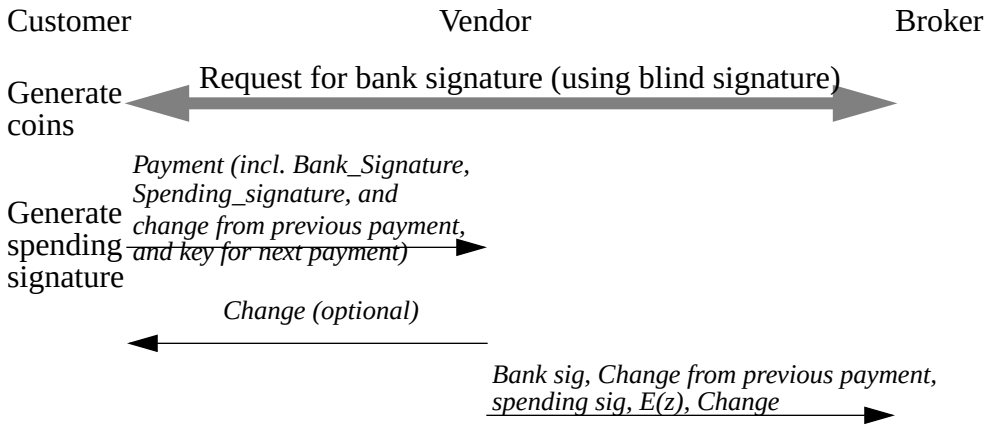

FIGURE 4. Summary of the Wenbo Scheme. The thick arrow refers to an interaction of a non-Wenbo Scheme.

6.2 Evaluation

6.2.1 Forgery Prevention

A bank signature authenticates and certifies the initial chain of coins, and a change certifies the unspent portion of the chain after each payment. A spending signature and a user certificate authenticate a payment and vouch it to the vendor.

6.2.2 Double Spending Detection

A customer's identity can be revealed if he attempts to double spend. This is because his user identity is inside the key for the next payment *($g(x_i)$)*, and this key is embedded in either a valid bank signature or a change from the previous payment. To generate a valid spending signature during a payment, the customer has to use the same key. When a customer double spends with valid spending signatures, a vendor or a broker finds that the bank signature and the coins are replayed and the spending signatures are different. In this case, the vendor or a broker can apply Schnorr's signature scheme to the spending signatures to reveal the customer's identity.

A customer can double spend with an invalid spending signature if a vendor colludes with him. In this case, the broker has to ask the certificate authority to reveal the encrypted customer's identity (*E(z)*).

When a vendor double deposits the payment, the spending signature is replayed because he cannot forge a customer's signature. Since the vendor is not anonymous to the broker at

redemption, the vendor can be caught easily. However, the customer's identity will not be revealed because the spending signature is the same.

6.2.3 Anonymity

The Wenbo Scheme satisfies the criterion of preserving a customer's identity from a broker, and thus it is anonymous (refer to Section 2.3 on page 6 for the definition of anonymity applied in this paper).

To prevent the bank from relating the coins to a customer, the scheme uses the blind signature technique during a withdrawal and a blinding function (g) to hide the customer identity in the bank signature. Moreover, it requires a vendor to encrypt the customer identity (decrypted only by a certificate authority) at a redemption.

6.2.4 Collusion

This subsection only describes the type of collusion that may not be solved. Other types of collusion are illustrated in the original paper ([8]).

In the Wenbo Scheme, a customer-vendor collusion means that a collusive vendor generates a valid change without receiving a valid spending signature from the customer. A valid spending signature means that the key ($g(x)$) in the spending signature must match the key in the previous change.

Suppose a customer wants to double spend his chain after $(k-1)$th payment by having V_E sign an extra change. After that, the customer pays V_k with V_{k-1}'s change, and $V_{k'}$ with V_E's change. When V_k and $V_{k'}$ turn in their payments, the broker detects the double spending, but he cannot use Schnorr's signature scheme to find the customer identity because the keys for the payments are different. Moreover, he cannot determine which party is colluding with the customer.

The broker has to rely on the key in V_{k-2}'s redemption to prove that V_E colludes with the customer. However, it is not known if V_{k-2} will ever redeem the payment. In other words, the case may never be solved. Figure 5 summarizes the collusion.

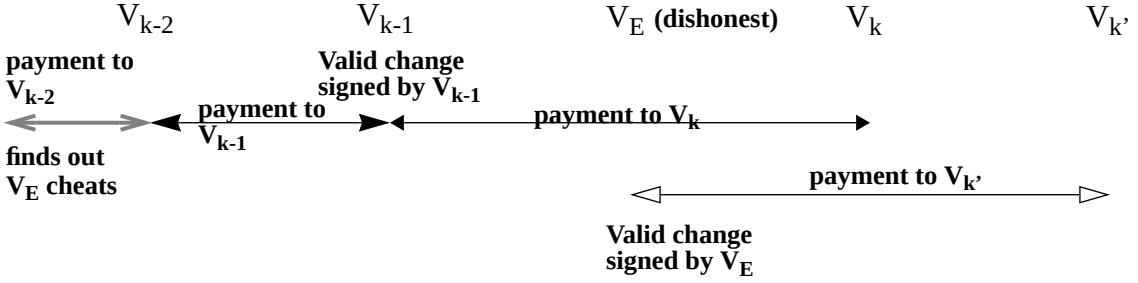

FIGURE 5. The customer asks V_E to sign a coin. V_E is malicious while others are honest. If V_{k-2} does not redeem the payment, the bank cannot find out which party is at fault.

6.2.5 Other Comments

Since Wenbo Scheme adopts the idea of authenticating a chain of hash values from Payword, some of its features are the same as Paywords. They are listed as follows:

- *Customers generate their own paywords.*
- *It uses the latest state for verification.*
- *Public key cryptography is required.*

Some features that are different from Payword's are as follows:

- *Customers need to pay in advance.* Unlike Payword, the Wenbo Scheme is debit-based to customers and brokers. A customer's account must be debited when the bank signature is issued to the coins because a broker cannot relate the coins to the customer at a redemption.
- *A long-term relationship is not required.* Because a chain is not committed to a particular vendor, there is no preference for a long-term customer-vendor relationship.
- *Chains need not be disjoint.* The hash function can be a counter. Unlike other payment schemes described in this paper, the hash function (h) in the Wenbo Scheme does not need to be one-way or collision-resistant since it requires a spending signature to prove that a purchase is initiated by the customer. In other words, the hash values of different chains need not be distinct.

Other nice features of the Wenbo Scheme are as follows:

- *Chains are universal.* Unlike payword chain that is customer-vendor specific, the chain of coins can be used to pay any vendors.
- *Customers are anonymous.* A customer identity is anonymous to a broker as long as the customer does not double spend and the vendor hides the customer identity at redemption.

Undesirable features of the Wenbo Scheme are as follows:

- *The Wenbo Scheme is more complicated.* Making the chain universal and having anonymity make the payment scheme more difficult to understand than the previous three payment protocols.
- *The number of signatures required grows with the number of payments.* A spending signature and change must be generated for each payment.

7.0 Conclusion

Four micropayment schemes have been evaluated in this paper: Millicent, MicroMint, Payword, and the Wenbo Scheme. They reduce computation costs by avoiding or amortizing the use of public key cryptography and by applying one-way and collision-resistant hash functions to improve both on-line and off-line performance. They also cut down the on-line storage and computation requirement by having an off-line broker. In addition, they allow a more transient relationship between customers and vendors. However, readers

should keep in mind that all four micropayment schemes require a secure channel to exchange secrets before a micropayment can take place.

Appendix A includes a summary of the features of the four payment schemes and compares their computation costs and storage requirements. Appendix B describes the details of Schnorr's signature scheme used in the Wenbo Scheme.

7.1 Acknowledgment

I would like to thank Keith Moore, Hal Abelson, Evan Kirshenbaum, and Tracy Sienknecht for their helpful feedback and Michelle Hogan and Justin Liu for editing the paper.

8.0 Appendix A - Summary Tables

TABLE 1. Summary of the features of the four payment schemes. C = user or customer. V= vendor. B = broker or bank.

	Denomination	**Produced by**	**Anonymity**	**Credit- or Debit-Based?**
Millicent	V- or B-specific scrip	V and/or B	No	Debit to C, V and B
Micro-Mint	Coins satisfying monthly criterion, U_id, and V_id	B	No	Debit to C and B, credit to V
Payword	U-V-specific chains of paywords	U	No	Credit
Wenbo Scheme	Chains of coins	U, certified by B	Yes, if no double spending	Debit to C and B, credit to V

TABLE 2. Summary of computation costs. Macropayment and double spending are excluded. Assume no frauds discovered during verification. C_U = customer's certificate.

	Before Payment	**For U, During Payment**	**V Verification**	**Redemption**
Millicent	(assume C has V's scrip for payment already)	1 hash/request	1 hash/request verification 1 hash/V's scrip verification 1 hash/change generation 1 hash/reply generation	None
Micro-Mint (Assumes a coin is a U-V-specific k-tuple	1 hash and storage/valid x-value generated k-1 subtraction to locate the groups of x-values satisfying U_id k searches to get the groups of x-values	1 hash/V_id k-1 subtraction k searches/coin built	**For each coin:** k hashes of x-values k comparison for monthly criterion 2(k-1) subtraction 1 hash/ U_id 1 hash/V_id	Same as V's
Payword	1 request for C_U 1 verification for C_U 1 user signature generation/commitment 1 hash/payword generation	(optional if not stored in advance) 1 hash/payword generation	1 signature verification/ commitment 1 signature verification/ C_U #(paywords-paid) hashes	commitment verification C_U verification #(paywords-redeemed) hashes
Wenbo Scheme	1 request for C_U 1 blind sig for bank signature generation/chain 1 hash/coin generation 1 key/payment for using Schnorr's scheme	1 spending signature generation/payment (optional if not stored in advance) 1 hash/coin generation	1 bank signature verification 1 spending signature verification using Schnorr's signature scheme 1 verification for change from V_{last} 1 signature generation for change #(coins-paid) hashes	1 bank signature verification previous V signature verification 1 change signature verification #(coins-paid) hashes

TABLE 3. Storage requirements of the four payment schemes. () means the item is optional, depending on the implementation of a protocol. CSK = customer-shared-key.

	Customer	**Vendor**	**Broker**
Millicent	Scrip purchased CSK	All V's valid scrip (CSK)	All B's valid scrip (V's scrip) CSK
MicroMint	Coins purchased	Valid received coins	All valid coins Customer purchase record
Payword	(Payword chain, commitments) C_U **For each chain:** n, i, w_n	All valid P_{last}'s and M's received	All valid P_{last}'s and M's redeemed
Wenbo Scheme	(coin chain) C_U **For each chain:** bank signature, last coin, n, i, change from V_{last} , and parameters for Schnorr's scheme	All valid signatures (and certificates)	All valid signatures (and certificates)

9.0 Appendix B — Schnorr's Signature Scheme

This section is for those who are interested in the details of the signature generation, the verification, and the mechanism for finding the user identity when the user double spends in the Wenbo Scheme.

9.1 Signature Generation

The customer chooses a user secret (*s*) to generate his public key (*v*). *p* is a large prime ($p > 2^{512}$), and *q* is another large prime such that $q|(p-1)$.

$$v = a^{-s} \bmod p$$
where
$$a^q = 1 \bmod p \qquad \textit{(EQ 20)}$$

To sign a message *m*, the user selects a one-time random number (*r*) such that

$$x = a^r \bmod p$$
$$e = H(m, x)$$
$$y = r + s \times e \bmod q \qquad \textit{(EQ 21)}$$

(e, y) is the signature pair for message m. In the Wenbo Scheme, this pair is called the *spending signature* of the coin (in this case, *m*). *H* is a one-way and collision-resistant hash function. *x* is the public key of *m*, and it is hashed into *g(x)*. *e*, *x*, *y,* and *v* are all public parameters.

9.2 Signature Verification

To verify a received message (*m'*) with its signature *(e, y)*, find *z*:

$$z = a^{y} v^{e} \bmod p \qquad \textit{(EQ 22)}$$

If $e = H(m', z)$ (i.e. *x* can be substituted with *z*), *m'* is verified.

9.3 Double Spending

If a user double spends, there will be two spending signatures for a common range of coins, *(e, y)* and *(e', y')*, and they both are generated by the same *x*. The user secret (*s*) can be found using the following relationship:

$$s = \frac{y - y'}{e - e'} \bmod q \qquad \textit{(EQ 23)}$$

10.0 References

1. D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10): 1030-1044, Oct. 1985. http://www.digicash.com/publish/bigbro.html.
2. DigiCash Inc. http://www.digicash.com/
3. S. Glassman, M. Manasse, et. al. *The Millicent Protocol for Inexpensive Electronic Commerce*. http://www.research.digital.com/SRC/millicent/papers/millicent-w3c4/millicent.html.
4. P. M. Hallam-Baker. *Micro Payment Transfer Protocol (MPTP), Version 0.1*. W3C Working Draft Nov. 22, 1995. http://www.w3.org/pub/WWW/TR/WD-mptp.
5. P. M. Hallam-Baker. *Electronic Payment Schemes*. http://www.w3.org/pub/WWW/Payments/roadmap.html.
6. Internet Keyed Payment Protocols (iKP). http://www.zurich.ibm.com:80/Technology/Security/extern/ecommerce/iKP_overview.htm
7. S. Low, N. F. Maxemchuk, and S. Paul. *Anonymous Credit Card*. Proceedings of the 2nd ACM Conference on Computer and Communication Security, Fairfax, Virginia, Nov. 2-4, 1994. ftp://ftp.research.att.com/dist/anoncc/anoncc.ps.Z.
8. W. Mao. *A Simple Cash Payment Technique for the Internet.* Proceedings of 1996 European Symposium on Research in Computer Science (ESORICS'96), Springer-Verlag, September 1996. http://wenbomao.hpl.hp.com/esorics96.ps.
9. W. Mao. Personal email addressed to the author on Aug. 23, 1996.
10. The NetBill Electronic Commerce Project at Carnegie Mellon University. http://www.ini.cmu.edu:80/NETBILL/
11. R. Rivest and A. Shamir. *PayWord and MicroMint: Two simple micropayment schemes.* May 7, 1996. http://theory.lcs.mit.edu/~rivest/RivestShamir-mpay.ps.

12. R. Rivest. *The MD5 Message Digest Algorithm*, Internet RFC 1321. http://theory.lcs.mit.edu/~rivest/Rivest-MD5.txt.

13. RSA Inc., “How large is a Modulus (Key) Should be Used in RSA?” *Frequently Asked Questions About Today’s Cryptography.* Version 3.0 http://www.rsa.com/rsalabs/newfaq/q12.html.

14. B. Schneier. Section 6.4, “Digital Cash,” *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, 1993.

15. B. Schneier. Section 14.5, “MD5,” *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, 1993.
This section has a missing + operation and misdefined <<<. Please refer to [17] for corrections.

16. E. Stefferud. Payment Systems for Electronic Commerce. *Berkeley Multimedia and Graphics Seminar.* University of California at Berkeley. Sep. 11, 1996.

17. H. P. Sun. “Lecture 6, 9-27-1995.” *6.915 Computer and Network Security.* Lecture by R. Rivest of Massachusetts Institute of Technology. http://theory.lcs.mit.edu/~rosario/6.915/lecture6.ps.

18. J. Thomas. “Lecture 17, November 7, 1995.” *6.915 Computer and Network Security.* Lecture by R. Rivest of Massachusetts Institute of Technology. http://theory.lcs.mit.edu/~rosario/6.915/lecture17.ps.