PayWord and MicroMint: Two simple micropayment schemes

Ronald L. Rivest* and Adi Shamir**

May 7, 1996

*MIT Laboratory for Computer Science
545 Technology Square, Cambridge, Mass. 02139

**Weizmann Institute of Science
Applied Mathematics Department
Rehovot, Israel

{rivest,shamir}@theory.lcs.mit.edu

1 Introduction

We present two simple micropayment schemes, "PayWord" and "MicroMint," for making small purchases over the Internet. We were inspired to work on this problem by DEC's "Millicent" scheme[10]. Surveys of some electronic payment schemes can be found in Hallam-Baker [6], Schneier[16], and Wayner[18].

Our main goal is to minimize the number of public-key operations required per payment, using hash operations instead whenever possible. As a rough guide, hash functions are about 100 times faster than RSA signature verification, and about 10,000 times faster than RSA signature generation: on a typical workstation, one can sign two messages per second, verify 200 signatures per second, and compute 20,000 hash function values per second.

To support micropayments, exceptional efficiency is required, otherwise the cost of the mechanism will exceed the value of the payments. As a consequence, our micropayment schemes are light-weight compared to full macropayment schemes. We "don't sweat the small stuff": a user who loses a micropayment is similar to someone who loses a nickel in a candy machine. Similarly, candy machines aren't built with expensive mechanisms for detecting forged coins, and yet they work well in practice, and the overall level of abuse is low. Large-scale and/or persistent fraud must be detected and eliminated, but if the scheme delivers a volume of payments to the right parties that is roughly correct, we're happy.

In our schemes the players are brokers, users, and vendors. Brokers authorize users to make micropayments to vendors, and redeem the payments collected by the vendors. While user-vendor relationships are transient, broker-user and broker-vendor relationships are long-term. In a typical transaction a vendor sells access to a World-Wide Web page for one cent. Since a user may access only a few pages before moving on, standard credit-card arrangements incur unacceptably high overheads.

The first scheme, "PayWord," is a credit-based scheme, based on chains of "paywords" (hash values). Similar chains have been previously proposed for different purposes: by Lamport [9] and Haller (in S/Key) for access control [7], and by Winternitz [11] as a one-time signature scheme. The application of this idea for micropayments has also been independently discovered by Anderson et al. [2] and by Pederson [14], as we learned after distributing the initial draft of this paper. We discuss these related proposals further in Section 5. The user authenticates a complete chain to the vendor with a single public-key signature, and then successively reveals each payword in the chain to the vendor to make micropayments. The incremental cost of a payment is thus one hash function computation per party. PayWord is optimized for sequences of micropayments, but is secure and flexible enough to support larger variable-value payments as well.

The second scheme, "MicroMint," was designed to eliminate public-key operations altogether. It has lower security but higher speed. It introduces a new paradigm of representing coins by k-way hash-function collisions. Just as for a real mint, a broker's "economy of scale" allows him to produce large quantities of such coins at very low cost per coin, while small-scale forgery attempts can only produce coins at a cost exceeding their value.

2 Generalities and Notation

We use public-key cryptography (e.g. RSA with a short public exponent). The public keys of the broker B, user U, and vendor V are denoted PK_B, PK_U, and PK_V, respectively; their secret keys are denoted SK_B, SK_U, and SK_V. A message M with its digital signature produced by secret key SK is denoted $\{M\}_{SK}$. This signature can be verified using the corresponding public key PK.

We let h denote a cryptographically strong hash function, such as MD5[15] or SHA[13]. The output (nominally 128 or 160 bits) may be truncated to shorter lengths as described later. The important property of h is its one-wayness and collision-resistance; a very large search should be required to find a single input producing a given output, or to find two inputs producing the same output. The input length may, in some cases, be equal to the output length.

3 PayWord

PayWord is credit-based. The user establishes an account with a broker, who issues her a digitally-signed PayWord Certificate containing the broker's name, the user's name and IP-address, the user's public key, the expiration date, and other information. The certificate has to be renewed by the broker (e.g. monthly), who will do so if the user's account is in good standing. This certificate authorizes the user to make Payword chains, and assures vendors that the user's paywords are redeemable by the broker. We assume in this paper that each payword is worth exactly one cent (this could be varied).

In our typical application, when U clicks on a link to a vendor V's non-free web page, his browser determines whether this is the first request to V that day. For a first request, U computes and signs a "commitment" to a new user-specific and vendor-specific chain of paywords w_1, w_2, $\ldots$, w_n. The user creates the payword chain in reverse order by picking the last payword w_n at random, and then computing

$$w_i = h(w_{i+1})$$

for $i = n-1$, $n-2$, $\ldots$, 0. Here w_0 is the root of the payword chain, and is not a payword itself. The commitment contains the root w_0, but not any payword w_i for $i > 0$. Then U provides this commitment and her certificate to V, who verifies their signatures.

The i-th payment (for $i = 1, 2, \ldots$) from U to V consists of the pair (w_i, i), which the vendor can verify using w_{i-1}. Each such payment requires no calculations by U, and only a single hash operation by V.

At the end of each day, V reports to B the last (highest-indexed) payment (w_l, l) received from each user that day, together with each corresponding commitment. B charges U's account l cents and pays l cents into V's account. (The broker might also charge subscription and/or transaction fees, which we ignore here.)

A fundamental design goal of PayWord is to minimize communication (particularly on-line communication) with the broker. We imagine that there will be only a few nationwide brokers; to prevent them from becoming a bottleneck, it is important that their computational burden be both reasonable and “off-line.” PayWord is an “off-line” scheme: V does not need to interact with B when U first contacts V, nor does V need to interact with B as each payment is made. Note that B does not even receive every payword spent, but only the *last* payword spent by each user each day at each vendor.

PayWord is thus extremely efficient when a user makes repeated requests from the same vendor, but is quite effective in any case. The public-key operations required by V are only signature verifications, which are relatively efficient. We note that Shamir's probabilistic signature screening techniques[17] can be used here to reduce the computational load on the vendor even further. Another application where PayWord is well-suited is the purchase of pay-per-view movies; the user can pay a few cents for each minute of viewing time.

This completes our overview; we now give some technical details.

3.1 User-Broker relationship and certificates

User U begins a relationship with broker B by requesting an account and a PayWord Certificate. She gives B over a secure authenticated channel: her credit-card number, her public key PK_U, and her “delivery address” A_U. Her aggregated PayWord charges will be charged to her credit-card account. Her delivery address is her Internet/email or her U.S. mail address; her certificate will only authorize payments by U for purchases to be delivered to A_U.

The user's certificate has an expiration date E. Certificates might expire monthly, for example. Users who don't pay their bills won't be issued new certificates.

The broker may also give other (possibly user-specific) information I_U in the certificate, such as: a certificate serial number, credit limits to be applied per vendor, information on how to contact the broker, broker/vendor terms and conditions, etc.

The user's certificate C_U thus has the form:

$$C_U = \{B, U, A_U, PK_U, E, I_U\}_{SK_B} .$$

The PayWord certificate is a statement by B to any vendor that B will redeem authentic paywords produced by U turned in before the given expiration date (plus a day's grace).

PayWord is not intended to provide user anonymity. Although certificates could contain user account numbers instead of user names, the inclusion of A_U effectively destroys U's anonymity. However, some privacy is provided, since there is no record kept as to which documents were purchased.

If U loses her secret key she should report it at once to B. Her liability should be limited in such cases, as it is for credit-card loss. However, if she does so repeatedly the broker may refuse her further service. The broker may also keep a “hot list” of certificates whose users have reported lost keys, or which are otherwise problematic.

As an alternative to hot-lists, one can use hash-chains in a different manner as proposed by Micali [12] to provide daily authentication of the user's certificate. The user's certificate would additionally contain the root w'_0 of a hash chain of length 31. On day $j-1$ of the month, the broker will send the user (e.g. via email) the value w'_j if and only if the user's account is still in good standing. Vendors will then demand of each user the appropriate w' value before accepting payment.

3.2 User-Vendor relationships and payments

User-vendor relationships are transient. A user may visit a web site, purchase ten pages, and then move on elsewhere.

Commitments

When U is about to contact a new vendor V, she computes a fresh payword chain w_1, $\ldots$, w_n with root w_0. Here n is chosen at the user's convenience; it could be ten or ten thousand. She then computes her commitment for that chain:

$$M = \{V, C_U, w_0, D, I_M\}_{SK_U} \ .$$

Here V identifies the vendor, C_U is U's certificate, w_0 is the root of the payword chain, D is the current date, and I_M is any additional information that may be desired (such as the length n of the payword chain). M is signed by U and given to V. (Since this signature is necessarily "on-line," as it contains the vendor's name, the user might consider using an "on-line/off-line" signature scheme[5].)

This commitment authorizes B to pay V for any of the paywords w_1, $\ldots$, w_n that V redeems with B before date D (plus a day's grace). Note that paywords are *vendor-specific* and *user-specific*; they are of no value to another vendor.

Note that U must sign a commitment for each vendor she pays. If she rapidly switches between vendors, the cost of doing so may become noticeable. However, this is PayWord's only significant computational requirement, and the security it provides makes PayWord usable even for larger "macropayments" (e.g. software selling at $19.99).

The vendor verifies U's signature on M and the broker's signature on C_U (contained within M), and checks expiration dates.

The vendor V should cache verified commitments until they expire at the end of the day. Otherwise, if he redeemed (and forgot) paywords received before the expiration date of the commitment, U could cheat V by replaying earlier commitments and paywords. (Actually, to defeat this attack, V need store only a short hash of each commitment he has reported to B already today.)

The user should preferably also cache her commitment until she believes that she is finished ordering information from V, or until the commitment expires. She can always generate a fresh commitment if she re-visits a vendor whose commitment she has deleted.

Payments

The user and vendor need to agree on the amount to be paid. In our exemplary application, the price of a web page is typically one cent, but could be some other amount. A web page should presumably be free if the user has already purchased it that day, and is just requesting it again because it was flushed from his cache of pages.

A payment P from U to V consists of a payword and its index:

$$P = (w_i, i) \ .$$

The payment is short: only twenty or thirty bytes long. (The first payment to V that day would normally accompany U's corresponding commitment; later payments are just the payword and its index, unless the previous chain is exhausted and a new chain must be committed to.) The payment is not signed by U, since it is self-authenticating (using the commitment).

The user spends her paywords in order: w_1 first, then w_2, and so on. If each payword is worth one cent, and each web page costs one cent, then she discloses w_i to V when she orders her i-th web page from V that day.

This leads to the PayWord payment policy: *for each commitment a vendor V is paid l cents, where (w_l, l) is the corresponding payment received with the largest index.* This means that V needs to store only one payment from each user: the one with the highest index. Once a user spends w_i, she can not spend w_j for $j < i$. The broker can confirm the value to be paid for w_l by determining how many applications of h are required to map w_l into w_0.

PayWord supports variable-size payments in a simple and natural manner. If U skips paywords, and gives w_7 after giving w_2, she is giving V a nickel instead of a penny. When U skips paywords, during verification V need only apply h a number of times proportional to the value of the payment made.

A payment does not specify what item it is payment for. The vendor may cheat U by sending him nothing, or the wrong item, in return. The user bears the risk of losing the payment, just as if he had put a penny in the mail. Vendors who so cheat their customers will be shunned. This risk can be moved to V, if V specifies payment *after* the document has been delivered. If U doesn't pay, V can notify B and/or refuse U further service. For micropayments, users and vendors might find either approach workable.

3.3 Vendor-Broker relationships and redemption

A vendor V needn't have a prior relationship with B, but does need to obtain PK_B in an authenticated manner, so he can authenticate certificates signed by B. He also needs to establish a way for B to pay V for paywords redeemed. (Brokers pay vendors by means outside the PayWord system.)

At the end of each day (or other suitable period), V sends B a redemption message giving, for each of B's users who have paid V that day (1) the commitment C_U received from U, (2) the last payment $P = (w_l, l)$ received from U.

The broker then needs to (1) verify each commitment received (he only needs to verify user signatures, since he can recognize his own certificates), including checking of dates, etc., and (2) verify each payment (w_l, l) (this requires l hash function applications). We assume that B normally honors all valid redemption requests.

Since hash function computations are cheap, and signature verifications are only moderately expensive, B's computational burden should be reasonable, particularly since it is more-or-less proportional to the payment volume he is supporting; B can charge transaction or subscription fees adequate to cover his computation costs. We also note that B never needs to respond in real-time; he can batch up his computations and perform them off-line overnight.

3.4 Efficiency

We summarize PayWord's computational and storage requirements:

- The broker needs to sign each user certificate, verify each user commitment, and perform one hash function application per payment. (All these computations are off-line.) The broker stores copies of user certificates and maintains accounts for users and vendors.
- The user needs to verify his certificates, sign each of his commitments, and perform one hash function application per payword committed to. (Only signing commitments is an on-line computation.) He needs to store his secret key SK_U, his active commitments, the corresponding payword chains, and his current position in each chain.
- The vendor verifies all certificates and commitments received, and performs one hash function application per payword received or skipped over. (All his computations are on-line.) The vendor needs to store all commitments and the last payment received per commitment each day.

3.5 Variations and Extensions

In one variation, $h(\cdot)$ is replaced by $h_s(\cdot) = h(s, \cdot)$, where s is a "salt" (random value) specified in the commitment. Salting may enable the use of faster hash functions or hash functions with a shorter output length (perhaps as short as 64–80 bits).

The value of each payword might be fixed at one cent, or might be specified in C_U or M. In a variation, M might authenticate several chains, whose paywords have different values (for penny paywords, nickel paywords, etc.).

The user name may also need to be specified in a payment if it is not clear from context. If U has more than one payword chain authorized for V, then the payment should specify which is relevant.

Paywords could be sold on a debit basis, rather than a credit basis, but only if the user interacts with the broker to produce each commitment: the certificate could require that the broker, rather than the user, sign each commitment. The broker can automatically refund the user for unused paywords, once the vendor has redeemed the paywords given to him.

In some cases, for macropayments, it might be useful to have the "commitment" act like an electronic credit card order or check without paywords being used at all. The commitment would specify the vendor and the amount to be paid.

The broker may specify in user certificates other terms and conditions to limit his risk. For example, B may limit the amount that U can spend per day at any vendor. Or, B may refuse payment if U's name is on B's "hot list" at the beginning of the day. (Vendors can down-load B's hot-list each morning.) Or, B may refuse to pay if U's total expenditures over all vendors exceeds a specified limit per day. This protects B from extensive liability if SK_U is stolen and abused. (Although again, since C_U only authorizes delivery to A_U, risk is reduced.) In these cases vendors share the risk with B.

Instead of using payword chains, another method we considered for improving efficiency was to have V *probabilistically* select payments for redemption. We couldn't make this idea work out, and leave this approach as an open problem.

4 MicroMint

MicroMint is designed to provide reasonable security at very low cost, and is optimized for unrelated low-value payments. MicroMint uses *no* public-key operations at all.

MicroMint "coins" are produced by a broker, who sells them to users. Users give these coins to vendors as payments. Vendors return coins to the broker in return for payment by other means.

A coin is a bit-string whose validity can be easily checked by anyone, but which is hard to produce. This is similar to the requirements for a public-key signature, whose complexity makes it an overkill for a transaction whose value is one cent. (PayWord uses signatures, but not on every transaction.)

MicroMint has the property that generating many coins is very much cheaper, per coin generated, than generating few coins. A large initial investment is required to generate the first coin, but then generating additional coins can be made progressively cheaper. This is similar to the economics for a regular mint, which invests in a lot of expensive machinery to make coins economically. (It makes no sense for a forger to produce coins in a way that costs more per coin produced than its value.)

The broker will typically issue new coins at the beginning of each month; the validity of these coins will expire at the end of the month. Unused coins are returned to the broker at the end of each month, and new coins can be purchased at the beginning of each month. Vendors can return the coins they collect to the broker at their convenience (e.g. at the end of each day).

We now describe the "basic" variant of MicroMint. Many extensions and variations are possible on this theme; we describe some of them in section 4.2.

Hash Function Collisions

MicroMint coins are represented by *hash function collisions*, for some specified one-way hash function h mapping m-bit strings x to n-bit strings y. We say that x is a pre-image of y if $h(x) = y$. A pair of distinct m-bit strings (x_1, x_2) is called a *(2-way) collision* if $h(x_1) = h(x_2) = y$, for some n-bit string y.

If h acts "randomly," the only way to produce even one acceptable 2-way collision is to hash about $\sqrt{2^n} = 2^{n/2}$ x-values and search for repeated outputs. This is essentially the "birthday paradox." (We ignore small constants in our analyses.)

Hashing c times as many x-values as are needed to produce the first collision results in approximately c^2 as many collisions, for $1 \leq c \leq 2^{n/2}$, so producing collisions can be done increasingly efficiently, per coin generated, once the threshold for finding collisions has been passed.

Coins as k-way collisions

A problem with 2-way collisions is that choosing a value of n small enough to make the broker's work feasible results in a situation where coins can be forged a bit too easily by an adversary. To raise the threshold further against would-be forgers, we propose using k-way collisions instead of 2-way collisions.

A k-way collision is a set of k distinct x-values x_1, x_2, ..., x_k that have the same hash value y. The number of x-values that must be examined before one expects to see the first k-way collision is then approximately $2^{n(k-1)/k}$. If one examines c times this many x-values, for $1 \leq c \leq 2^{n/k}$, one expects to see about c^k k-way collisions. Choosing $k > 2$ has the dual effect of delaying the threshold where the first collision is seen, and also accelerating the rate of collision generation, once the threshold is passed.

We thus let a k-way collision $(x_1, \ldots, x_k)$ represent a coin. The validity of this coin can be easily verified by anyone by checking that the x_i's are distinct and that

$$h(x_1) = h(x_2) = \cdots = h(x_k) = y$$

for some n-string y.

Minting coins

The process of computing $h(x) = y$ is analogous to tossing a ball (x) at random into one of 2^n bins; the bin that ball x ends up in is the one with index y. A coin is thus a set of k balls that have been tossed into the same bin. Getting k balls into the same bin requires tossing a substantial number of balls altogether, since balls can not be "aimed" at a particular bin. To mint coins, the broker will create 2^n bins, toss approximately $k2^n$ balls, and create one coin from each bin that now contains at least k balls. With this choice of parameters each ball has a chance of roughly 1/2 of being part of a coin.

Whenever one of the 2^n bins has k or more balls in it, k of those balls can be extracted to form a coin. Note that if a bin has more than k balls in it, the broker can in principle extract k-subsets in multiple ways to produce several coins. However, an adversary who obtains two different coins from the same bin could combine them to produce multiple new coins. Therefore, we recommend that *a MicroMint broker should produce at most one coin from each bin.* Following this rule also simplifies the Broker's task of detecting multiply-spent coins, since he needs to allocate a table of only 2^n bits to indicate whether a coin with a particular n-bit hash value has already been redeemed.

A small problem in this basic picture, however, is that computation is much cheaper than storage. The number of balls that can be tossed into bins in a month-long computation far exceeds both the number of balls that can be memorized on a reasonable number of hard disks and the number of coins that the broker might realistically need to mint. One could attempt to balance the computation and memory requirements by utilizing a very slow hash algorithm, such as DES iterated many times. Unfortunately, this approach also slows down the verification process.

A better approach, which we adopt, is to make most balls unusable for the purpose of minting coins. To do so, we say that a ball is "good" if the high-order bits of the hash value y have a value z specified by the broker. More precisely, let $n = t + u$ for some specified nonnegative integers t and u. If the high-order t bits of y are equal to the specified value z then the value y is called "good, " and the low-order u bits of y determine the index of the bin into which the (good) ball x is tossed. (General x values are referred to merely as "balls," and those that are not good can be thought of as having been conceptually tossed into nonexistent virtual bins that are "out of range.")

A proper choice of t enables us to balance the computational and storage requirements of the broker, without slowing down the verification process. It slows down the generation process by a factor of 2^t, while limiting the storage requirements of the broker to a small multiple of the number of coins to be generated. The broker thus tosses approximately $k2^n$ balls, memorizes about $k2^u$ good balls that he tosses into the 2^u bins, and generates from them approximately $(1/2) \cdot 2^u$ valid coins.

Remark: We note that with standard hash functions, such as MD5 and DES, the number of ouput bits produced may exceed the number n of bits specified in the broker's parameters. A suitable hash function for the broker can be obtained by discarding all but the low-order n bits of the standard hash function output. This discarding of bits other than the low-order n bits is a different process than that of specifying a particular value for the high-order t bits out of the n that was described above.

A detailed scenario

Here is a detailed sketch of how a typical broker might proceed to choose parameters for his minting operating for a given month. The calculations are approximate (values are typically rounded to the nearest power of two), but instructive; they can be easily modified for other assumptions.

The broker will invest in substantial hardware that gives him a computational advantage over would-be forgers, and run this hardware continuously for a month to compute coins valid for the next month. This hardware is likely to include many special-purpose chips for computing h efficiently.

We suppose that the broker wishes to have a net profit of \$1 million per month (approximately 2^{27} cents/month). He charges a brokerage fee of 10%. That is, for every coin worth one cent that he sells, he only gives the vendor 0.9 cents when it is redeemed. Thus, the broker needs to sell one billion coins per month (approximately 2^{30} coins/month) to collect his \$1M fee. If an average user buys 2500 (\$25.00) coins per month, he will need to have a customer base of 500,000 customers.

The broker chooses $k = 4$; a coin will be a good 4-way collision.

To create 2^{30} coins, the broker chooses $u = 31$, so that he creates an array of 2^{31} (approximately two billion) bins, each of which can hold up to 4 x-values that hash to an n-bit value that is the concantenation of a fixed t-bit pattern z and the u-bit index of the bin.

The broker will toss an average of 4 balls into each bin. That is, the broker will generate $4 \cdot 2^{31} = 2^{33}$ (approximately eight billion) x-values that produce good y-values. When he does so, the probability that a bin then contains 4 or more x-values (and thus can yield a coin) is about 1/2. (Using a Poisson approximation, it can be calculated that the correct value is approximately 0.56.) Since each of the 2^{31} bins produces a coin with probability 1/2, the number of coins produced is 2^{30}, as desired.

In order to maximize his advantage over an adversary who wishes to forge coins, the broker invests in special-purpose hardware that allows him to compute hash values very quickly. This will allow him to choose a relatively large value of t, so that good hash values are relatively rare. This increases the work factor for an adversary (and for the broker) by a factor of 2^t. The broker chooses his hash function h as the low-order n bits of the encryption of some fixed value v_0 with key x under the Data Encryption Standard (DES):

$$h(x) = [DES_x(v_0)]_{1\ldots n} .$$

The broker purchases a number of field-programmable gate array (FPGA) chips, each of which is capable of hashing approximately 2^{25} (approximately 30 million) x-values per second. (See [3].) Each such chip costs about \$200; we estimate that the broker's actual cost per chip might be closer to \$400 per chip when engineering, support, and associated hardware are also considered. The broker purchases 2^8 (= 256) of these chips, which costs him about \$100,000. These chips can collectively hash 2^{33} (approximately 8.6 billion) values per second. Since there are roughly 2^{21} (two million) seconds in a month, they can hash about 2^{54} (approximately 18 million billion) values per month.

Based on these estimates the broker chooses $n = 52$ and $t = 21$ and runs his minting operation for one month. Of the $k2^n = 2^{54}$ hash values computed, only one in 2^{21} will be good, so that approximately 2^{33} good x-values are found, as necessary to produce 2^{30} coins.

Storing a good $(x, h(x))$ pair takes less than 16 bytes. The total storage required for all good pairs is less than 2^{37} bytes (128 Gigabytes). Using standard magnetic hard disk technology costing approximately \$300 per Gigabyte, the total cost for storage is less than \$40,000. The total cost for the broker's hardware is thus less than \$150,000, which is less than 15% of the first month's profit.

Rather than actually writing each pair into a randomly-accessible bin, the broker can write the 2^{33} good pairs sequentially to the disk array, and then sort them into increasing order by y value, to determine which are in the same bin. With a reasonable sorting algorithm, the sorting time should be under one day.

Selling coins

Towards the end of each month, the broker begins selling coins to users for the next month. At the beginning of each month, B reveals the new validity criterion for coins to be used that month. Such sales can either be on a debit basis or a credit basis, since B will be able to recognize coins when they are returned to him for redemption. In a typical purchase, a user might buy \$25.00 worth of coins (2500 coins), and charge the purchase to his credit card. The broker keeps a record of which coins each user bought. Unused coins are returned to the broker at the end of each month.

Making payments

Each time a user purchases a web page, he gives the vendor a previously unspent coin $(x_1, x_2, \ldots, x_k)$. (This might be handled automatically by the user's web browser when the user clicks on a link that has a declared fee.) The vendor verifies that it is indeed a good k-way collision by computing $h(x_i)$ for $1 \le i \le k$, and checking that the values are equal and good. Note that while the broker's minting process was intentionally slowed down by a factor of 2^t, the vendor's task of verifying a coin remains extremely efficient, requiring only k hash computations and a few comparisons (in our proposed scenario, $k = 4$).

Redemptions

The vendor returns the coins he has collected to the broker at the end of each day. The broker checks each coin to see if it has been previously returned, and if not, pays the vendor one cent (minus his brokerage fee) for each coin. We propose that if the broker receives a specific coin more than once, he does not pay more than once. Which vendor gets paid can be decided arbitrarily or randomly by the broker. This may penalize vendors, but eliminates any financial motivation a vendor might have had to cheat by redistributing coins he has collected to other vendors.

4.1 Security Properties

We distinguish between small-scale attacks and large-scale attacks. We believe that users and vendors will have little motivation to cheat in order to gain only a few cents; even if they do, the consequences are of no great concern. This is similar to the way ordinary change is handled: many people don't even bother to count their change following a purchase. Our security mechanisms are thus primarily designed to discourage large-scale attacks, such as massive forgery or persistent double-spending.

Forgery

Small-scale forgery is too expensive to be of interest to an adversary: with the recommended choice of $k = 4$, $n = 54$, and $u = 31$, the generation of the first forged coin requires about 2^{45} hash operations. Since a standard work-station can perform only 2^{14} hash operations per second, a typical user will need 2^{31} seconds (about 80 years) to generate just one forged coin on his workstation.

Large-scale forgery can be detected and countered as follows:

- All forged coins automatically become invalid at the end of the month.
- Forged coins can not be generated until after the broker announces the new monthly coin validity criterion at the beginning of the month.
- The use of hidden predicates (described below) gives a finer time resolution for rejecting forged coins without affecting the validity of legal coins already in circulation.
- The broker can detect the presence of a forger by noting when he receives coins correspondings to bins that he did not produce coins from. This works well in our scenario since only about half of the bins produce coins. To implement this the broker need only work with a bit-array having one bit per bin.
- The broker can at any time declare the current period to be over, recall all coins for the current period, and issue new coins using a new validation procedure.
- The broker can simultaneously generate coins for several future months in a longer computation, as described below; this makes it harder for a forger to catch up with the broker.

Theft of coins

If theft of coins is judged to be a problem during initial distribution to users or during redemption by vendors, it is easy to transmit coins in encrypted form during these operations.

User/broker and vendor/broker relationships are relatively stable, and long-term encryption keys can be arranged between them.

To protect coins as they are being transferred over the Internet from user to vendor, one can of course use public-key techniques to provide secure communication. However, in keeping with our desire to minimize or eliminate public-key operations, we propose below another mechanism, which makes coins user-specific. This does not require public-key cryptography, and makes it harder to re-use stolen coins.

Another concern is that two vendors may collude so that both attempt to redeem the same coins. The recommended solution is that a broker redeem a coin at most once, as discussed earlier. Since this may penalize honest vendors who receive stolen coins, we can make coins vendor-specific as well as user-specific, as described below.

Double-spending

Since the MicroMint scheme is not anonymous, the broker can detect a doubly-spent coin, and can identify which vendors he received the two instances from. He also knows which user the coin was issued to. With the vendors' honest cooperation, he can also identify which users spent each instance of that coin. Based on all this information, the broker can keep track of how many doubly-spent coins are asssociated with each user and vendor. A large-scale cheater (either user or vendor) can be identified by the large number of duplicate coins associated with his purchases or redemptions; the broker can then drop a large-scale cheater from the system. A small-scale cheater may be hard to identify, but, due to the low value of individual coins, it is not so important if he escapes identification.

MicroMint does not provide any mechanism for preventing purely malicious framing (with no financial benefit to the framer). We believe that the known mechanisms for protecting against such behavior are too cumbersome for a light-weight micropayment scheme. Since MicroMint does not use real digital signatures, it may be hard to legally prove who is guilty of duplicating coins. Thus, a broker will not be able to pursue a cheater in court, but can always drop a suspected cheater from the system.

4.2 Variations

User-specific coins

We describe two proposals for making coins that are user-specific in a way that can be easily checked by vendors. Such coins, if stolen, are of no value to most other users. This greatly reduces the motivation for theft of coins.

In the first proposal, the broker splits the users into "groups," and gives each user coins whose validity depends on the identity of the group. For example, the broker can give user U coins that satisfy the additional condition $h'(x_1, x_2, \ldots, x_k) = h'(U)$, where hash function h' produces short (e.g. 16-bit) output values that indicate U's group. A vendor can easily check this condition, and reject a coin that is not tendered by a member of the correct group.

The problem with this approach is that if the groups are too large, then a thief can easily find users of the appropriate group who might be willing to buy stolen coins. On the other hand, if the groups are too small (e.g. by placing each user is in his own group), the broker may be forced to precompute a large excess of coins, just to ensure that he has a large enough supply to satisfy each user's unpredictable needs.

In the second proposal, we generalize the notion of a "collision" to more complicated combinatorial structures. Formally, a coin $(x_1, \ldots, x_k)$ will be valid for a user U if the images $y_1 = h(x_1)$, $y_2 = h(x_2)$, ..., $y_k = h(x_k)$ satisfy the condition

$$y_{i+1} - y_i = d_i \pmod{2^u}$$

for $i = 1, 2, \ldots, k-1$, where

$$(d_1, d_2, \ldots, d_{k-1}) = h'(U)$$

for a suitable auxiliary hash function h'. (The original proposal for representing coins as collisions can be viewed as the special case where all the distances d_i's between the k bins are zero.)

To mint coins of this form, the broker fills up most of his bins by randomly tossing balls into them, except that now it is not necessary to have more than one ball per bin. We emphasize that this pre-computation is not user-specific, and the broker does not need to have any prior knowledge of the number of coins that will be requested by each user, since each good ball can be used in a coin for *any* user. After this lengthy pre-computation, the broker can quickly create a coin for any user U by

- Computing $(d_1, \ldots, d_{k-1}) = h'(U)$.
- Picking a random bin index y_1. (This bin should have been previously unused as a y_1 for another coin, so that y_1 can be used as the "identity" of the coin when the broker uses a bit-array to determine which coins have already been redeemed.)
- Computing $y_{i+1} = y_i + d_i \pmod{2^u}$ for $i = 1, 2, \ldots, k-1$,
- Taking a ball x_1 out of bin y_1, and taking a copy of one ball out of each bin y_2, ..., y_k. (If any bin y_i is empty, start over with a new y_1.) Note that balls may be re-used in this scheme.
- Producing the ordered k-tuple $(x_1, \ldots, x_k)$ as the output coin.

A convenient feature of this scheme is that it is easy to produce a large number of coins for a given user even when the broker's storage device is a magnetic disk with a relatively slow seek time. The idea is based on the observation that if the y_1 values for successive coins are consecutive, then so also will be the y_i values for each i, $1 < i \leq k$. Therefore, a request for 2500 new coins with $k = 4$ requires only four disk seeks, rather than $10,000$ seeks: at 10 milliseconds per seek, this reduces the total seek time from 100 seconds to only 40 milliseconds.

Note that in principle coins produced for different users could re-use the same ball x_i. Conceivably, someone could forge a new coin by combining pieces of other coins he has seen. However, he is unlikely to achieve much success by this route unless he sees balls from a significant fraction of all the bins. For example, suppose that there are 2^{31} bins, of which the forger has seen a fraction 2^{-10} (i.e., he has collected 2^{21} balls from coins spent by other users). Then the expected number of coins he can piece together from these balls that satisfy the condition of being a good coin for himself is only $2^{31}(2^{-10})^3 = 2$. (Even if he had 1000 customers for these coins, he would expect to make only 2000 coins total, or two coins per customer on the average.) Thus, we are not too concerned about this sort of "cut-and-paste" forgery.

Vendor-specific coins

To further reduce the likelihood that coins will be stolen, the user can give coins to vendors in such a way that each coin can be redeemed only by a small fraction of the vendors. This technique makes a stolen coin less desirable, since it is unlikely to be accepted by a vendor other than the one where it was originally spent. The additional check of validity can be carried out both by the vendor and by the broker. (Having vendor-specific coins is also a major feature of the Millicent [10] scheme.)

The obvious difficulty is that neither the broker nor the user can predict ahead of time which vendors the user will patronize, and it is unreasonable to force the user to purchase in advance coins specific for each possible vendor. Millicent adopts the alternative strategy whereby the user must contact the broker in real-time whenever the user needs coins for a new vendor. (He also needs to contact the broker to return excess unused coins that are specific to that vendor.) We can overcome these problems with an extension of the user-specific scheme described above, in which the user purchases a block of "successive" MicroMint coins.

Intuitively, the idea is the following. Choose a value v (e.g. 1024) less than u. Let a u-bit bin-index y be divided into a $u-v$-bit upper part y' and a v-bit lower part y''. We consider that y' specifies a "superbin" index and that y'' specifies a bin within that superbin. A user now purchases balls in bulk and makes his own coins. He purchases balls by the superbin, obtaining 2^v balls per superbin with one ball in each bin of the superbin. He buy k superbins of balls for 2^v cents. A coin from user U is valid for redemption by vendor V if:

$$y'_{i+1} = y'_i + d'_i \pmod{2^{u-v}} \text{ for i} = 1, \ldots, \text{k} - 1,$$

and

$$y''_{i+1} = y''_i + d''_i \pmod{2^v} \text{ for i} = 1, \ldots, \text{k} - 1,$$

where

$$h'(U) = (d'_1, \ldots, d'_{k-1})$$

and

$$h''(V) = (d''_1, \ldots, d''_{k-1}) \ .$$

The broker chooses the next available superbin as the first superbin to give the user; the other superbins are then uniquely determined by the differences $\{d'_i\}$ defined by the user's identity and the choice of the first superbin. Analogously, to make a coin for a particular vendor the user chooses a ball from the next bin from his first superbin, and must use balls from bins in the other superbins that are then uniquely determined by the differences $\{d''_i\}$ defined by the vendor's identity and the choice of the first bin. Note that balls from the first superbin are used only once, to permit detection of double-spending, whereas balls from the other superbins may appear more than once (in coins paid to different vendors), or not at all. It may be difficult for a broker to create superbins that are perfectly full even if he throws more balls. He might sell superbins that are almost full, but then a user may have difficulty producing some coins for some vendors. To compensate, the broker can reduce the price by one cent for each empty bin sold.

Simultaneously generating balls for multiple months

Our major line of defense against large-scale forgery is the fact that the broker can compute coins in advance, whereas a forgery attempt can only be started once the new validity condition for the current month is announced. We now describe a technique whereby computing the balls for a single month's coins takes eight months, but the broker doesn't fall behind because he can generate balls for eight future months concurrently. The forger will thus have the dual problems of starting late and being too slow, even if he uses the same computational resources as the real broker.

In this method, the broker changes the monthly validity criterion, not by changing the hash function h, but by announcing each month a new value z such that ball x is good when the high-order t bits of $h(x)$ are equal to z. The broker randomly and secretly chooses in advance the values z that will be used for each of the next eight months. Tossing a ball still means performing one hash function computation, but the tossed ball is potentially "good" for any of the next eight months, and it is trivial for the broker to determine if this is the case. In contrast, the forger only knows the current value of z, and can not afford to memorize all the balls he tosses, since memory is relatively expensive and only a tiny fraction (e.g., 2^{-21} in our running example) of the balls are considered "good" at any given month.

We now describe a convenient way of carrying out this calculation. Assume that at the beginning of the month j, the broker has all of the balls needed for month j, $7/8$ of the balls needed for month $j+1$, $6/8$ of the balls needed for month $j+2$, ..., and $1/8$ of the balls needed in for month $j+7$. During month j, the broker tosses balls by randomly picking x values, calculating $y = h(x)$, and checking whether the top-most t bits of y are equal to any of the z values to be used in months $j+1, \ldots, j+8$. To slow the rate at which he generates good balls for each upcoming month, he increases n and t each by three. After the month-long computation, we expect him to have all the coins he needs for month $j+1$, $7/8$ of the coins he needs for month $j+2$, and so on; this is the desired "steady-state" situation. The broker needs four times as much storage to hold the balls generated for future months, but balls for future months can be temporarily stored on inexpensive magnetic tapes because he doesn't need to respond quickly to user requests for those coins yet.

Hidden Predicates

The "hidden predicate" technique for defeating forgers works as follows. We choose $m > n$, and require each m-bit pre-image to satisfy a number of hidden predicates. The hidden predicates should be such that generating pre-images satisfying the predicates is easy (if you know the predicate). To generate an x_i, one can pick its last n bits randomly, and define the j-th bit of x_i, for $j = m-n, \ldots, 1$, to be the j-th hidden predicate applied to bits $j+1, \ldots, m$ of x_i. The hidden predicates must be balanced and difficult to learn from random examples. Suggestions of hard-to-learn predicates exist in the learning-theory literature. For example the parity/majority functions of Blum et al.[4] (which are the exclusive-or of some of the input bits together with the majority function on a disjoint set of input bits) are interesting, although slightly more complicated functions may be appropriate in this application when word lengths are short. With $m - n = 32$, the broker can have one hidden predicate for each day of the month. He could reveal a new predicate each day, and ask vendors to check that the coins they receive satisfy these predicates (otherwise the coins will not be accepted by the broker). This would not affect the validity of legitimate coins already in circulation, but makes forgery extremely difficult, since the would-be forger would have to discard much of his precomputation work as each new predicate is revealed. We feel that such techniques are strongly advisable in MicroMint.

Other Extensions

Peter Wayner (private communication) has suggested a variation on MicroMint in which coins of different values are distinguished by publicly-known predicates on the x-values.

5 Relationship to Other Micropayment Schemes

In this section we compare our proposals to the Millicent[10], NetBill [1], NetCard [2], and Pederson [14] micropayment schemes.

NetBill offers a number of advanced features (such as electronic purchase orders and encryption of purchased information), but it is relative expensive: digital signatures are heavily used and the NetBill server is involved in each payment.

Millicent uses hash functions extensively, but the broker must be on-line whenever the user wishes to interact with a new vendor. The user buys vendor-specific scrip from the broker. For applications such as web browsing, where new user-vendor relationships are continually being created, Millicent can place a heavy real-time burden on the broker. Compared to Millicent, both PayWord and MicroMint enable the user to generate vendor-specific "scrip" without any interaction with the broker, and without the overhead required in returning unused vendor-specific scrip. Also, PayWord is a credit rather than debit scheme.

Anderson, Manifavas, and Sutherland [2] have developed a micropayment system, "NetCard," which is very similar to PayWord in that it uses chains of hash values with a digitally signed root. (The way hash chains are created differs in a minor way.) However, in their proposal, it is the bank rather than the user who prepares the chain and signs the root, which adds to the overall burden of the bank. This approach prevents the user from creating new chains, although a NetCard user could spend a single chain many times. Compared to PayWord, NetCard is debit-based, rather than credit-based. We have heard that a patent has been applied for on the NetCard system.

Torben Pedersen outlines a micropayment proposal[14] that is also based on hash chains. His motivating application was for incremental payment of telephone charges. His paper does not provide much detail on many points (e.g. whether the system is credit or debit-based, how to handle exceptions, whether chains are vendor-specific, and other auxiliary security-related matters). The CAFE project has filed for a patent on what we believe is an elaboration of Pedersen's idea. (The details off the CAFE scheme are not available to us.)

Similarly following Pedersen's exposition, the iKP developers Hauser, Steiner, and Waidner have independently adopted a similar approach [8].

6 Conclusions and Discussion

We have presented two new micropayment schemes which are exceptionally economical in terms of the number of public-key operations employed. Furthermore, both schemes are *off-line* from the broker's point of view.

References

[1] The NetBill Electronic Commerce Project, 1995. `http://www.ini.cmu/NETBILL/home.html`.

[2] Ross Anderson, Harry Manifavas, and Chris Sutherland. A practical electronic cash system, 1995. Available from author: `Ross.Anderson@cl.cam.ac.uk`.

[3] Matt Blaze, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. Minimal key lengths for symmetric ciphers to provide adequate commercial security: A report by an ad hoc group of cryptographers and computer scientists, January 1996. Available at `http://www.bsa.org`.

[4] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Proc. CRYPTO 93*, pages 278–291. Springer, 1994. Lecture Notes in Computer Science No. 773.

[5] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 263–277. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[6] Phillip Hallam-Baker. W3C payments resources, 1995. `http://www.w3.org/hypertext/WWW/Payments/overview.html`.

[7] Neil M. Haller. The S/KEY one-time password system. In *ISOC*, 1994.

[8] Ralf Hauser, Michael Steiner, and Michael Waidner. Micro-Payments based on iKP, December 17, 1995. Available from authors. `sti@zurich.ibm.com`.

[9] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–771, November 1981.

[10] Mark S. Manasse. Millicent (electronic microcommerce), 1995. `http://www.research.digital.com/SRC/personal/Mark_Manasse/uncommon/ucom.html`.

[11] Ralph C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 218–238. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[12] Silvio Micali. Efficient certificate revocation. Technical Report TM-542b, MIT Laboratory for Computer Science, March 22, 1996.

[13] National Institute of Standards and Technology (NIST). *FIPS Publication 180: Secure Hash Standard (SHS)*, May 11, 1993.

[14] Torben P. Pedersen. Electronic payments of small amounts. Technical Report DAIMI PB-495, Aarhus University, Computer Science Department, Århus, Denmark, August 1995.

[15] Ronald L. Rivest. The MD5 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1321.

[16] Bruce Schneier. *Applied Cryptography (Second Edition)*. John Wiley & Sons, 1996.

[17] Adi Shamir. Fast signature screening. CRYPTO '95 rump session talk; to appear in RSA Laboratories' *CryptoBytes*.

[18] Peter Wayner. *Digital Cash: Commerce on the Net.* Academic Press, 1996.