PayMe!

A customer-oriented efficient micropayment scheme.

Philip Lisiecki and Yoav Yerushalmi

6.915 project

Fall 1995

Abstract: We propose a micropayments scheme for use on the internet for purchases of online documents and services. This scheme is non-anonymous, and does not require a continually-available banker. It differs from other schemes in that it gives a new list of priorities for online transactions. First, the customer is being protected more than the vendor. In the scenario described, a vendor can profit from not providing useful service to a thousand customers all at once before being caught, while the customer stands to gain very little, per transaction, from cheating. The other important change is our requirement for low-latency for the customer, and high bandwidth for the vendor. The customer should not have to wait too long for every page on the web for example, and the vendor, in order to make a profit, is going to have to be able to handle a very large number of requests daily. In this paper, we propose a protocol and algorithms that satisfy these requirements.

Table of Contents

1.0 Introduction

The Internet has made it possible for many people to access a large volume of information around the world. It has also opened up the possibility of a new market, where vendors can charge customers very small fees for the right to browse documents. Ultimately, this will be at the pay-per-view level, where the cost may be measured in fractions of cents. To make this possible, an infrastructure has to be laid out. There are several requirements for this system (to be covered in section 2), and based on these, we have designed a protocol using a combination of a modification on the Rabin[6] signature scheme (proposed by Shamir) and S/KEY[3] (as in PayWord[5] proposed by Shamir and Rivest).

2.0 System Requirements

PayMe! assumes a centralized bank, multiple customers, and several vendors. For the purposes of discussion, we will use access to web pages as examples, but the protocol works fine for any service. The vendor makes offers to the customer, and when the customer wishes to purchase, he accepts, receives the document, and pays for it. The following principles are the basis for the protocol chosen, and the algorithms recommended for use:

- The customer is at the top of the priority list. It is least beneficial for the customer to cheat in this system, since he will get at most free access to pages, which will be tracked back to that user, who can then be thrown off the system, without a great loss to any single vendor. The customer must have low latency between selecting a page and viewing it [2]. This means that although some computation is permitted, intensive computations should be done before the selection is made wherever possible, and caching wherever possible is also recommended.
- The vendor, in order to make a profit, will need to process many requests per day. If every such request required intensive computation, the vendor would be unable to keep up with the pace of transactions, or would have to invest heavily in fast computers, both of which would cut into the profits [2]. The vendor also has periods of peak usage and of low usage, so it would be helpful if computations were carried out in the low-usage times whenever possible. Since every transaction is for a very small amount of money, the vendor can afford to let through a false request with controlled probability, if this speeds up processing of requests. This observation lets us use a scheme proposed by Shamir which can probabilistically verify a signature in time much faster than the time required to go through full verification, and yet where complete verification is also possible. This scheme is based on the difficulty of computing square roots without knowing the factorization (Rabin's signature scheme with a small modification). Finally, since it is expected that a customer will purchase multiple pages from the vendor, it will be nice if this could be exploited to allow for less computation on subsequent requests—as in the PayWord scheme. However, it would be advantageous if whatever scheme were used did not require the vendor to use too much memory or to keep a large customer database.

- The bank is necessary to transfer the actual money from the account of the customer to the account of the vendor. Also, it is the only entity that is familiar with all the other entities involved in a transaction and can authenticate for the two by signing messages using public keys known and certified by the bank. The bank can afford the fast computers necessary to do whatever processing is required. It should be allowed to go off-line for some periods of time, though it will be assumed that the bank can maintain a presence on the network for most of the time (at least the time necessary for reporting of transactions). The bank needs to be able to handle the high volume of requests that are submitted daily. Any amount of pre-processing that can be done at the vendor end cheaply is helpful.
- Anonymity is not required. This scheme is designed for purchases which can be easily traced via packet sniffing even if the payment and bartering were done anonymously. The extra overhead involved in making every transaction anonymous, as well as the high volume of generated transactions expected make anonymity a problem.
- Minor errors are tolerable, as long as they can be traced, and are monetarily insignificant (if one in a million pages are accidentally given away free, that is acceptable, however, it must be possible to trace a cheater so that he can be removed from the system).

Those are the basic requirements for our system, and our protocol attempts to deal with those.

3.0 The System

Given the above requirements, we can now design a system which meets those requirements. This is not the only possible system, and one can optimize differently, but it is one we thought is appropriate. First, the algorithms being used will be discussed, and then the protocol will be discussed, using the algorithms as building blocks.

3.1 Signatures

In several places in our protocol, we will need to sign and verify signatures. We have selected a specific modification of the Rabin scheme for digital signatures. The modification allows for fast probabilistic verification of the signature. There will never be a false reject, but there can be false accepts.

In Rabin

$$s^2 = m \pmod n$$

where s is the signature, m is the message, and n is a product of two large primes whose factorization is the secret information. Note that the message must be a quadratic residue for this scheme to be useful. In our modification to Rabin, we still use the same formula, but calculate a new value, c, which is the value we shall use as the signature:

$$s^2 = m + cn$$

s is still $\sqrt{m} \pmod{n}$. c and s are of relatively the same magnitude, and directly related. Figuring out c should be as hard as figuring out s. Given a c, however, it is easy to verify whether the resulting value is indeed a perfect square. At this point, we can verify using the above formula, and know without error whether the signature is valid. In order to speed things up, however, we choose to do our calculations in a much smaller field. This will allow for the possibility of errors, but can be repeated several times to increase confidence in validity. The algorithm is as follows:

1. Choose a small random prime p.
2. Take the given signature c, the message m, and the public modulus n calculate their values taken $mod\ p$. This will yield c_p, m_p, and n_p.
3. Make sure that $m_p + c_p n_p$ is QR in Z_p.
4. Repeat as many times as is felt necessary, or as load allows.

Without proving the algorithm secure, we can argue its strength. Knowledge of c is directly related to knowledge of s^2 over the integers, which implies knowledge of s. Finding a value s (or two distinct s's that are not additive inverses, if the message s^2 were found by squaring some known s) is equivalent to the knowledge of the factorization of n, which is assumed hard. Therefore, transmission of c is useful as a signature. A passive adversary, having seen c and m, will learn nothing of the factorization of n. An active adversary can still exploit the chosen plaintext attack on this scheme, and so it is very important that nobody sign any random messages they receive. Furthermore, an adversary pretending to be the signer will need to find either a c such that $cn + m$ is a perfect square, or is a quadratic residue in the field of all small primes that the verifier is likely to choose. Both of these are considerably hard for a useful m.

However, we need to map message m to a QR, so that this scheme will be useful. The following is the method we propose for mapping any m uniquely to a QR in $Z_n{}^*$:

1. Pick n such that $\left(\frac{2}{n}\right) = -1$. Make sure all your m's are $1 \le m \le \frac{n}{8}$.
2. If $\left(\frac{m \cdot 1}{n}\right) = 1$ then $m_1 = m \cdot 10$, else $m_1 = m \cdot 1$. Now, $\left(\frac{m_1}{n}\right) = 1$
3. If this m_1 is QR, then $m_2 = m_1$, else $m_2 = -m_1$. We now use m_2 as the message, which has a one-to-one mapping with m, but is a QR in Z_n^*.

Finally, the verifier can be reasonably sure that what he has received was generated by the signer. The distribution of QR and QNR mapped mod Z_p is about even. If the adversary were to randomly choose a signature randomly, the verifier will have a 50% chance of noticing in one pass. Even though the signature message does not change with every pass, the field in which we are checking for quadratic residuousity does, and so we can increase the probability of catching a false signature by randomly choosing new moduli.

Ultimately, since calculations in smaller fields are much faster than calculations in large ones, several passes through this are likely to be faster than one squaring operation where large primes are involved. Also, since the primes can be chosen very small, instead of doing calculations to find out if the resulting value is a QR, the program can just use a pre-computed lookup table.

3.2 Hashing

Further on in our protocol, we shall also be using a hashing scheme. It is useful for calculating MAC's, and because hashing is MUCH faster than signing, a chain of hashed values with an unknown starting value (like in S/KEY and PayWord) will be used, which relies on a hashing function. We place very few requirements on the hashing scheme, except that it be very fast to compute. We will assume that the hashing scheme used is such that given a hash value, it is hard to find a value that will hash to that.

Once again, many schemes are applicable. For our protocol, a scheme such as MD5[4] is appropriate, simply because it is very widely accepted and available, and although it has not been around as long as some other schemes, it is considered reliable enough by others for us to use it in our protocol, while accepting the possibility that this might have to change.

For a description and an implementation of MD5, please refer to RFC1321.

4.0 The protocol

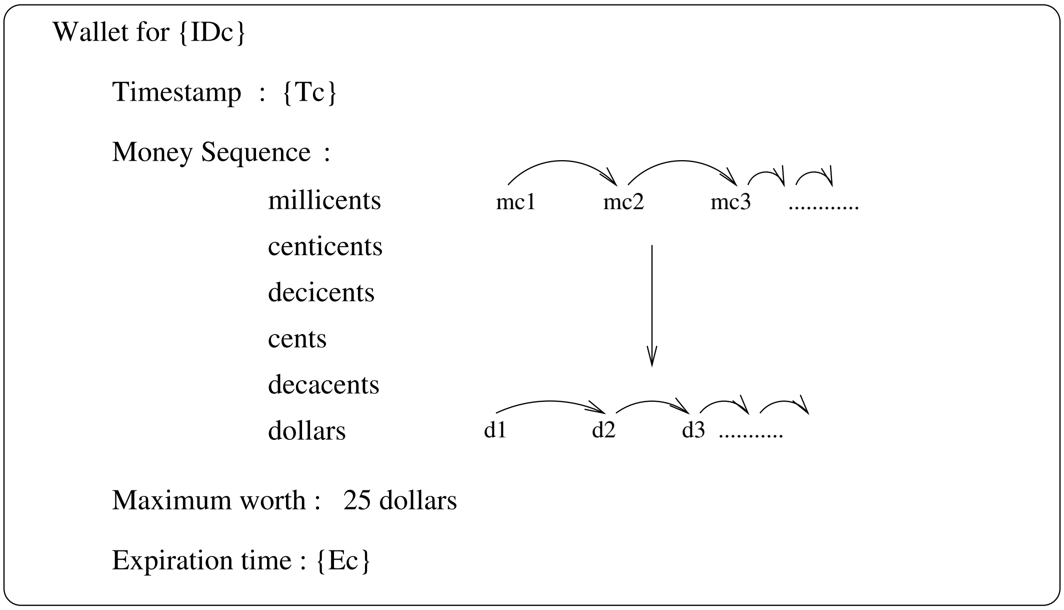
With the above basic tools, we can now describe the actual protocol, and analyze it both for security and efficiency. We also want to see how well it actually satisfies our system requirements.

4.1 Protocol description

The PayMe! Protocol operates according to the following rules:

1. In the beginning of the day (or when he has run out of wallets), a customer creates "wallets." A wallet is composed of a timestamp, several sequences of locked "coins" (encoded as in S/KEY), an identifier of the customer, and a maximum value the wallet is worth, as illustrated in the following diagram:

FIGURE 1. A sample wallet for PayMe!



NOTE: In the sequences of coins, the only thing actually stored as a number in the wallet is the last value of the chain (however many values customer actually knows). The customer can later reveal each element in the sequence one by one in order to pay. Furthermore, an customer's identifier contains the following information: the account number, an IP address or related information about where to send or receive information to or from, and also the public key of the customer.

2. The customer submits the wallets for the bank to sign. The bank verifies the wallet does match the customer's ID, that the customer has a valid account, and that it is happy with the maximum worth of each of the wallets it has received (this is important, since there is no way for the banker to know how long the chain of coins is in each sequence). It also makes sure the timestamp is reasonable. If the banker is happy, he signs the wallet, and sends it back. The customer receives:

$$\langle ID_c, T_c, money - sequence, max - worth \rangle_{K_B}$$

3. In the course of the day, the customer would like to purchase a page from a vendor. If he has not dealt with the vendor already that day, he will take a wallet he has not used yet, and ask the vendor for a timestamp (unique identifier which can expire and will be used as a handle for the customer). He will then append the vendor's name and timestamp to the wallet, sign it, and send it to the vendor:

$$\langle \langle ID_c, T_c, money - sequence, max - worth \rangle_{K_B}, ID_V, T_V \rangle_{K_C}$$

4. The vendor receives the wallet and extras, and verifies the entire message as well as the wallet (probabilistically if there is too much load at the moment). He checks the timestamp to make sure it is not in use and exceeds the customer's last vendor-timestamp. The vendor is now ready to sell to the customer. The vendor saves the wallet so that it may be referenced as money is spent from that wallet.
5. When the customer is ready to purchase, he sends a request for a page, which includes the name of the document, the vendor timestamp he has (the handle), and his own ID.

$$name, T_V, ID_C$$

6. If the vendor believes the wallet he has can pay for the document requested (by comparing the maximum value of the wallet and amount of money still left to spend with the price being quoted), he transmits the document/page to the customer whose ID he received.
7. If the customer is satisfied with what he has received, he then releases enough coins to pay for the document, as was agreed to previously. He only needs to release one number per coin value, since the rest of the chain can be figured from that. So to release 15¢, he can release the 15th cent coin from where he is currently, or release the next 10¢ coin, and five 1¢ coins.
8. The vendor then verifies that the customer has indeed sent him valid cash by following the hash chain. The vendor should then mark the fact that the wallet has been used to pay that much money, and subtract that from the total amount of cash in the wallet.
9. This process is repeated until the wallet has run out of money, or is close to doing so. At this point, the customer sends a new wallet if he wishes to continue transacting, and the vendor puts the old wallet aside, remembering the last hash value of the coins in each chain.
10. At the end of the day, the vendor then transmits the wallets to the bank, sorted first by customer, and then by their unique identifiers. Each wallet is sent with the amount of money spent in each denomination and a matching value for the coins in the chains. It is important that this transaction be atomic, or at least able to be restarted at whatever point it was terminated. The protocol must also ensure that the two parties are indeed the vendor and the bank. Simple flow-control, message authentication codes, and signatures can be used for these purposes.
11. The bank makes sure that the wallets are arriving in increasing identifier order per customer (not non-decreasing, as we want to avoid duplicates). He verifies the wallets one by one (the coins and the signatures) and transfers appropriate cash from the customer account to the vendor account. Again, this can be done probabilistically, but as soon as an error is encountered, it is suggested that he verify everything from that customer and vendor, just in case. Only vendor-timestamps in a certain range specified by the vendor will be accepted. The lower bound of this range must be greater than the upper bound from the previous wallet-transmission cycle, guaranteeing that a given customer's wallet signature is good for one and only one wallet transmission.

This protocol achieves the basic goals set out, and also has the following interesting properties:

- There is no actual encryption happening here. While the payment protocol uses no encryption, vendors might choose to use protocols such as S/HTTP or SSL in order to avoid sniffers from gaining free pages. The only cryptographically interesting aspects of the system are hashing and signing, both of which are much easier to export from the United States than encryption.
- This protocol is flexible in its choice of algorithms. Nothing here depends on a particular side-effect of any of the algorithms described. All that is required is an efficient hashing scheme, and a probabilistic signature scheme (which can be avoided or used only with extreme caution).
- The choice of a modified Rabin scheme for this protocol opens up the possibility of the keys being revealed through a chosen message attack. However, no signatures here are actually made blindly. In order to exploit a chosen message attack, one must find a number whose square is a message that contains all the required ID's, timestamps, and other information. This is not necessarily simple and should be made as difficult as possible. Updating keys on a regular basis can lower the risk and amount of time people have to generate such chosen messages.

4.2 Storage Needs

The customer must store his public and private keys, all of the signed wallets that have not yet been used as well as the corresponding starting hash values. Additionally, the customer needs to keep track of the currently active wallets for each vendor, along with their starting hash values and vendor-timestamps.

The vendor needs ready access to the open wallet for each customer since the last update with the bank. Once a wallet is closed, i.e., once the customer begins spending from a new wallet, the old wallet may be put aside until the bank update occurs. By requiring the new vendor-timestamp of a new wallet to be greater than the one currently being closed, the wallets that have been set aside need not be consulted when a customer begins spending from a new wallet.

The bank, of course, needs to store all of the account balances, customers' public keys, and vendors' highest redeemed vendor-timestamps. Any other information stored is only for accounting and reporting to the customer. (Additional detailed logs may also be kept to allow a bank to defend itself against accusations of being corrupt; see Section 4.3.5, "Malicious Bank," on page 11 for more details.)

4.3 Security Analysis of the Protocol

In order to analyze the security of the protocol, one must first have a threat model. We analyze the case of a passive adversary, an active adversary, a malicious customer, a malicious vendor, and of course, a malicious bank. A secret key or a locked coin in a wallet can be discovered. A forgery of a wallet or a coin is possible. Replay is a possibility, as well as modification of any of the above in order to cheat any or all of the participants. For each case, we analyze what can be done, and see whether it is a problem. Note that a customer will only use each wallet once, so the revealed values that unlock coins in the wallet are not useful in any context other than the transaction requested by the customer.

4.3.1 Passive Adversary

Nothing in this protocol is encrypted or hidden. A passive adversary will learn everything that is transmitted on the network. The only secret information that can be learned by the adversary is the next unlocking of a coin, by which point the merchant will have already received the same thing. Gaining a signed wallet is useless, since he doesn't know how to unlock the coins. Reusing a wallet after seeing some unlocked coins is impossible, since he can't sign the vendor-id and identifier to the wallet he has. All he can do is learn non-secret information. This system is secure against a passive adversary.

4.3.2 Active Adversary as an Outsider

The active adversary can do many things to disrupt the operations of this protocol. He can disconnect any or all of the connections or alter data. By being able to control the wires, he can get some useful information, though the net value of what can be gained is small, and he is likely to be noticed and stopped quickly. He cannot fake a signature, so he can't pretend to be the customer until the customer has done all the authentication. He can't pretend to be the vendor successfully, since the customer doesn't pay until he receives what he wants and likes it. Finally, pretending to be the bank will be of little use, since the adversary can't sign wallets. So, the only useful attack he can attempt is to wait until the customer has sent the signed wallet to the vendor, and then, when the customer attempts to pay, intercept the payment and attempt to use that knowledge later. If the vendor accepts the fact that the customer has not paid for the page, and lets the customer proceed, then the adversary can use the coins he's grabbed to request something he can pay for with those. This is not very useful, nor is it very costly to any participant. Furthermore, this will be noticed, and hopefully tracked down.

The best an active adversary can do is to request a page on behalf of a customer, and then refuse to pay for it. This can be prevented, however, by using an underlying protocol that makes the insertion of spurious requests on the stream difficult.

4.3.3 Malicious Customer

A malicious customer can reuse a wallet, but that is to his disadvantage (he is risking having somebody know how to unlock coins by noticing the use of the previous wallet). Furthermore, the bank will accept the reused wallet as another wallet if the vendor-timestamp is not the same. If the customer tries to reuse a vendor-timestamp, the vendor refuses the wallet.

Another possibility is for the customer to refuse to pay, and he will in fact get pages for free this way. However, the vendors will know this is happening, and can easily complain to the bank. If this is a regular event, the customer can be removed from the system.

4.3.4 Malicious Vendor

This protocol provides very good protection against a malicious vendor. First of all, the vendor may not steal from the customer in the customer-vendor interaction, since the customer does not actually pay until after the service is delivered and since the vendor has no information about future coins until the coin hash preimages are actually released by the customer.

Furthermore, the vendor may not easily steal from the customer or the bank when redeeming wallets. Only the correct vendor can redeem a wallet, since the wallet is signed by the customer with that vendor's name and vendor-timestamp. A single vendor cannot redeem a wallet multiple times either, though, since the vendor-timestamp signed into each wallet by the customer must be presented in increasing order. The vendor-timestamp windows also prevent a vendor from submitting a wallet in two different redemption transactions. Thus, a wallet may only be redeemed once, and only by the correct vendor.

4.3.5 Malicious Bank

If banks are distrusted, with additional storage on the bank's computer, this protocol can be made secure against a malicious bank by logging all redemption transactions on the bank's computer. (These logs need not be accessible on-line, only when a complaint is filed against the bank.) Note that this protocol uses the bank as the public-key authority. So, to protect against a malicious bank, a trusted third-party public-key authority must also be used.

A vendor can prove that a transaction occurred by presenting the wallet again. (This also requires additional long-term, off-line storage for the vendor.) The bank can check the logged transactions to make sure that the transaction was in fact not already credited to the vendor's account. If it was not, the bank has been caught. If it was, the vendor has been caught.

The bank can prove that a customer authorized a transaction by presenting the redemption transaction. This transaction contains a customer's signature on the wallet, along with coin hash values that only the customer could have determined, since those values are linked back to the customer-signed wallet.

A customer with incorrect charges can request that the bank prove that the charges were made by demanding that the bank show the redeemed wallet. If the bank cannot prove that the transaction occurred, the customer has effectively proven that the transaction did not occur (at least under United States banking laws).

4.4 Performance Issues

4.4.1 Off-line processing

All of the processing involving the bank is done off-line, with respect to customer-vendor interactions. The customer obtains new, signed wallets well before beginning a transaction with a vendor, and the vendor redeems wallets in a batch process that can be performed completely independently from normal sales. This allows the bank to handle the load of customers and vendors at whatever rate it can, so long as work does not build up at the bank from day to day.

4.4.2 Signatures

Generally, public-key signatures are very expensive to perform, and very expensive to verify. The modified Rabin scheme simplifies verification, but does not aid (or hurt significantly) the running time of generation of signatures. Hashing is generally much, much quicker than public-key operations.

The use of signatures has been minimized throughout the protocol, while preserving the goal of off-line wallet creation and redemption. The bank must digitally sign each wallet that a customer presents. Note, however, that the time to sign a wallet, including processing any backlog on the bank's server, does not slow down the user, as long as the user keeps a few spare wallets around. The user must make one digital signature once for each vendor that must be paid (per wallet used with that vendor). Once this signature is complete, this customer and vendor may do business with no signature overhead until the wallet runs out or the vendor redeems the wallet. The vendor must verify the customer's digital signature. But, this can be done probabilistically, and need only be performed when the wallet is initially exchanged. Note that this places most of the burden of cryptography on the customer, who is presumably dealing with a small number of vendors, while the vendor is doing a quick, simple verification of a great number of customers' signatures. Once wallets have been exchanged, only quick hash operations need to be performed by the vendor to verify subsequent purchases.

The order of growth of a typical public-key signature such as with Rabin, is $O(k^3)$, where k is the number of bits in the key, assuming that the signature can be performed with a single exponentiation operation, which is the case if n is chosen correctly. The verification of a standard Rabin signature requires squaring the signature, which is an $O(k^2)$ operation.

The modified Rabin scheme should still require $O(k^3)$ to produce a signature, since the square root must still be performed, followed by an application of the extended-Euclid algorithm to find the value c. But, a round of verification on the signature takes only $O(k)$ time, since the primes used to verify are assumed to be of small, fixed size. To verify, the modulus n, the signature c, and the message m must be considered mod some small prime p, so three k-bit over small-fixed-bit modulo operations need to be performed. These can each be performed in $O(k)$ time, and the remaining operations in the verifications are all $O(1)$ operations in the field Z_p^*, yielding a composite running time bound of $O(k)$.

4.4.3 Timings

To provide some insight as to the relative speeds of the various algorithms, we implemented standard Rabin, Modified-Robin, and RSA signature verification algorithms using the GNU C++ Integer class which supports arbitrary-length integer representations and run on a Pentium 90 running NetBSD 1.1.

TABLE 1. Performance of Various Signature Verification Scheme Implementations (times are measured in microseconds)

	256-bit key	512-bit key	2048-bit key
Rabin	650	2050	28000
Mod-Robin (valid)	3650	3750	39400
Mod-Rabin (forged)	1450	1500	20900
Optimized Mod-Robin (valid)	300	500	1850
Optimized Mod-Rabin (forged)	150	250	750
RSA	26100	168000	9520000

Notice that the Rabin implementation is faster than the Modified-Rabin implementation (with 5 verification rounds) for all key sizes tested, which range from being too small to be considered secure, to being larger than is generally considered secure, given the present state of factoring algorithms and availability of computational power. The growth of Rabin appears to be about $O(k^2)$, as was predicted. Modified-Rabin starts from behind and is rapidly catching up to regular Rabin. If even larger keys were to be used, the Modified-Rabin algorithm would presumably surpass Rabin. The other standard implementation we did was of RSA, which turned out to be significantly slower than the Rabin variants. In all of these cases, there are likely to be a few optimizations that will boost performance fairly significantly.

In the case of Modified-Rabin, however, there is an enormous potential for optimization. Notice that the first step of a round of verification is to take the three parameters of the signature and computing their values mod p. The GNU C++ Integer class does not take advantage of p's very small size in these modulus computations. The entire modulus operator can be done in $O(k)$ time by considering the input from the most to the least significant part. In fact, by performing some precomputation, the modulus operation can be reduced to the equivalent of a convolution of the bytes of the input number with a fixed set of numbers, mod Z_p. The final multiplication, addition, and QR check can be implemented as a simple single-word multiply and single-word addition followed by a table lookup on the resulting value. Notice that the resulting "Optimized Mod-Rabin" implementation is rather significantly faster than any of the other signature algorithms, and exhibits the $O(k)$ growth that was expected. Our test machine would be able to perform about 500 verifications per second of signatures with a 2048-bit key.

5.0 Conclusion

The PayMe! protocol is a simple, reliable protocol to facilitate small electronic payments. It uses minimal cryptography, and yet is fairly secure and safe. Using a probabilistic signatures and off-line bank interaction, customer latencies are kept low while vendor throughput is kept high.

References

[1] Phillip Hallam-Baker. Micro Payment Transfer Protocol. Preliminary version, November, 1995.

[2] Phillip Hallam-Baker. Interview, November 1995.

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

[4] Ronald L. Rivest. The MD5 Message-Digest Algorithm. Request for Comments: 1321.

[5] Ronald L. Rivest and Adi Shamir: PayWord and MicroMint: Two simple micropayment schemes. (Submitted to Eurocrypt '96), 1995.

[6] Adi Shamir. Interview, November 1995.

Appendix:

source code used

common.h

```
#ifndef _COMMON_H
#define _COMMON_H

#include <Integer.h>

#define SIZE_OF_KEYS 1024

typedef struct {
  Integer msg;      // message
  Integer sig;      // signature
} signed_message;

#endif
```

quick_mod.h

```
#ifndef _QUICK_MOD_H
#define _QUICK_MOD_H

#include “common.h”
#include “utils.h”
#include “rabin.h”
extern “C” {
  #include <sys/time.h>
}

extern void init_quick_sig ( int tries );
extern int check_sig_quick ( rabin_public_key, signed_message,
                            struct timeval *thing );

#endif
```

quick_mod.C

```
#include <Integer.h>
#include “common.h”
#include “rabin.h”
#include “quick_mod.h”
extern “C” {
  #include <sys/time.h>
}

typedef unsigned char BYTE;

BYTE *prepare_mod ( int *bigprep, int bignumbytes, int modulus )
{
  int i;
  int x = 1;
  int value = 1;
  BYTE *residues = new BYTE[modulus];
  if ( !residues )
    return NULL;
  for ( i = 0; i < bignumbytes; i++ ) {
    bigprep[i] = value;
    value <<= 8;
    value %= modulus;
  }
  for ( i = 0; i < modulus; i ++ )
    residues[i] = 0;
  for ( i = 0; i < modulus; i ++ )
    residues[(i*i)%modulus] = 1;
  return residues;
}

int quick_mod ( BYTE *bignum, int *bigprep, int bignumbytes, int modulus )
```

```
{
  int i;
  int result = 0;
  for ( i = 0; i < bignumbytes; i ++ )
    /* Shouldn't overflow for #'s not too long.  :-) */
    result += bigprep[i] * bignum[i];
  return result % modulus;
}

int check_sig ( BYTE *n, BYTE *m, BYTE *c, int modulus,
               int *modprep, BYTE *residueprep, int bignumbytes )
{
  int np, mp, cp;
  np = quick_mod ( n, modprep, bignumbytes, modulus );
  mp = quick_mod ( m, modprep, bignumbytes, modulus );
  cp = quick_mod ( c, modprep, bignumbytes, modulus );
  return residueprep[(mp+np*cp)%modulus];
}

int prepare_for_check_sig ( Integer a, BYTE *bignum, int bignumbytes )
{
  int i, j, k, x;
  for ( i = j = 0; i < bignumbytes; i ++ ) {
    x = 0;
    for ( k = 0; k < 8; j++, k++ ) {
      int bit = testbit(a, j) ? 1 : 0;
      x += bit << k;
    }
    bignum[i] = x;
  }
}

#define BYTES (((SIZE_OF_KEYS*2)+31)/8)
#define MAXTRIES 100

int   modprep[MAXTRIES][BYTES];
BYTE *resprep[MAXTRIES];
int   num_tries = 0;
int   moduli[MAXTRIES];

void init_quick_sig ( int tries )
{
  int i;
  if ( tries > MAXTRIES ) {
    cout << "Internal try limit exceeded!\n";
    return;
  }
  if ( num_tries ) {
    for ( i = 0; i < num_tries; i ++ )
      delete resprep[i];
  }
  num_tries = tries;
  for ( i = 0; i < tries; i ++ ) {
    int modulus = generate_prime(TEMP_PRIME_SIZE).as_long();
    moduli[i] = modulus;
    resprep[i] = prepare_mod ( modprep[i], BYTES, modulus );
    if ( !resprep[i] ) {
      cout << "Allocated failed.\n";
      exit ( -1 );
    }
  }
}

int check_sig_quick ( rabin_public_key pubkey, signed_message m,
                     struct timeval *thing )
{
/*
  Integer a;
  BYTE aa[BYTES];
  a = 1;
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
  a = 256;
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
  a = 65536;
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
  a = 1000000000L;
```

```
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
  a *= 1234567890L;
  a += 75847334L;
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
  a *= 8483247L;
  a += 349234L;
  prepare_for_check_sig ( a, aa, BYTES );
  cout << "quick_mod: " << quick_mod ( aa, modprep[0], BYTES, moduli[0] ) << "\n";
  cout << "      mod: " << (a) % (moduli[0]) << "\n";
*/
  int  result;
  BYTE nn[BYTES], mm[BYTES], cc[BYTES];
  prepare_for_check_sig ( pubkey.n, nn, BYTES );
  prepare_for_check_sig ( m.msg,    mm, BYTES );
  prepare_for_check_sig ( m.sig,    cc, BYTES );
  gettimeofday(thing, NULL);
  for ( int i = 0; i < num_tries; i ++ ) {
/*
    cout << "Modulus: " << moduli[i] << "\n";
    cout << "resprep: "
      << (char)(resprep[i][0]+'0') << (char)(resprep[i][1]+'0') << (char)(resprep[i][2]+'0') <<
(char)(resprep[i][3]+'0')
        << (char)(resprep[i][4]+'0') << (char)(resprep[i][5]+'0') << (char)(resprep[i][6]+'0') <<
(char)(resprep[i][7]+'0')
          << "\n";
    cout << "Mine: " << quick_mod ( nn, modprep[i], BYTES, moduli[i] );
    cout << " theirs: " << (pubkey.n) % moduli[i];
    cout << " mod: " << moduli[i] << "\n";
    cout << "Theirs: " << (pubkey.n) % moduli[i] << " "
      << (m.msg) % moduli[i] << " "
        << (m.sig) % moduli[i] << "\n";
*/
    result = check_sig ( nn, mm, cc, moduli[i], modprep[i], resprep[i],
                        BYTES );
    if ( !result )
      return 0;
  }
  return 1;
}
```

rabin.h

```
#ifndef _RABIN_H
#define _RABIN_H

#include "common.h"
#include "utils.h"

#define TEMP_PRIME_SIZE 15  // number of bits for modified rabin prime
#define NUMBER_TRIES    5  // number of passes to use by default.

typedef struct {
  Integer p, q, n;
} rabin_private_key;

typedef struct {
  Integer n;
} rabin_public_key;

extern void       gen_rabin_key    (rabin_private_key *);
extern void       rabin_sign       (rabin_private_key, signed_message *);
extern int        rabin_check      (rabin_public_key, signed_message);
extern void       mod_rabin_sign   (rabin_private_key, signed_message *);
extern int        mod_rabin_check  (rabin_public_key, signed_message, int);

#endif
```

rabin.C

```
#include "rabin.h"

void gen_rabin_key(rabin_private_key *k)
/* This function generates a secret rabin key, which can
```

```
 * then be used to derive the public info. For now, it creates
 *  a key pair s.t p = q = 3 mod 4
 */
{
  do
    k->p = generate_prime(SIZE_OF_KEYS);
  while ((k->p % 4) != 3);
  do
    k->q = generate_prime(SIZE_OF_KEYS);
  while ((k->q % 4) != 3);
  k->n = k->p * k->q;
}

void rabin_sign(rabin_private_key k, signed_message *message)
/* This function takes a message, and modifies it in such a
 * way so that it becomes signed. It takes a message with
 * an invalid signature, and a key k, and puts a valid signature
 * in the message. NOTE, it is assumed that the thing to be signed
 * is a quadratic residue for this implementation. Please check.
 *   it uses a standard rabin signature.
 */
{
  // check that this is indeed a QRn
  if ((legendre(message->msg, k.p) != 1) ||
      (legendre(message->msg, k.q) != 1)) {
        cerr << "sign_message: got a non quadratic residue message!!\n";
        return;
      }
  message->sig = sqrt2(message->msg, k.p, k.q);  //  set s to be the sqrt, which exists.
}

int rabin_check(rabin_public_key k, signed_message m)
/* This function attempts to figure out whether a signature for a message
     is valid for the user who'se public key is k
 */
{
 if (exp_mod(m.sig, 2, k.n) == m.msg)
    return 1;
  else
    return 0;
}

void mod_rabin_sign(rabin_private_key k, signed_message *message)
/* This will generate the signature of the type
   s^2 = m + cn, and returns a c s.t. s^2=m mod n;
 */
{
  Integer s;
  // check that this is indeed a QRn
  if ((legendre(message->msg, k.p) != 1) ||
      (legendre(message->msg, k.q) != 1)) {
        cerr << "sign_message: got a non quadratic residue message!!\n";
        return;
      }
  s = sqrt2(message->msg, k.p, k.q);  //  set s to be the sqrt, which exists.
  message->sig = ((pow(s,2) - message->msg) / k.n);
}

int mod_rabin_check(rabin_public_key k, signed_message m, int times_to_try)
/* This function should attempt to probabilistically check a message
      for a valid signature. It will do many passes on the same algorithm
      */
{
  long temp_prime, number;

  while (times_to_try--) {
    temp_prime = generate_prime(TEMP_PRIME_SIZE).as_long();
    number = ((m.msg%temp_prime + ((k.n%temp_prime) * (m.sig%temp_prime))) % temp_prime).as_long();
    if (legendre(number, temp_prime) != 1){
      return 0;  ///  AHA!!  not a valid signature !!
      }
  }
  return 1; //  couldn't find an invalid case.
}
```

rsa.h

```
#ifndef _RSA_H
#define _RSA_H
```

```
#include "common.h"

typedef struct {
  Integer p, q, n;
  Integer d, e;
} rsa_private_key;

typedef struct {
  Integer n;
  Integer e;
} rsa_public_key;

extern void      gen_rsa_key    (rsa_private_key *);
extern void      rsa_sign       (rsa_private_key, signed_message *);
extern int       rsa_check      (rsa_public_key, signed_message);

#endif
```

rsa.C

```
#include "utils.h"
#include "rsa.h"

void      gen_rsa_key    (rsa_private_key *k)
/* generate a valid RSA key pair */
{
  Integer phi;

  k->p = generate_prime(SIZE_OF_KEYS);
  k->q = generate_prime(SIZE_OF_KEYS);
  k->n = k->p * k->q;
  phi = (k->p -1) * (k->q -1);
  do {
    k->e = generate_random(SIZE_OF_KEYS/2);
  } while (gcd(k->e, phi) != 1);
  k->d = inverse(k->e, phi);
}

void      rsa_sign       (rsa_private_key k, signed_message *m)
/* sign a message */
{
  m->sig = exp_mod(m->msg, k.d, k.n);
}

int       rsa_check      (rsa_public_key k, signed_message m)
/* verify a signature */
{
  if (exp_mod(m.sig, k.e, k.n) == m.msg)
    return 1;
  else
    return 0;
}
```

signature.C

```
#include <iostream.h>
extern "C" {
  #include <sys/time.h>
}
#include "utils.h"
#include "rabin.h"
#include "rsa.h"
#include "quick_mod.h"

#define REPEAT 5

int main(), valid();
void test();

int main(){

  init();
  test();
}

int valid()
{
return 1;
```

```
}

void test()
/* Yep.. this tests the routines to see if they do the right thing */
{
  Integer            temp;
  signed_message     m;
  rabin_public_key   rabin_p;
  rabin_private_key  rabin_s;
  rsa_public_key     rsa_p;
  rsa_private_key    rsa_s;
  struct timeval     thing, thing2;
  long               secs, usecs, total_usecs;
  int                repetitions;

  cout << "Inverse      --\n";
  ((inverse(5, 11) == 9) || (inverse(5, 11) == 9) ||
   (inverse(5, 11) == 9) || (inverse(5, 11) == 9)) ? cout << "\t\tokay\n" : cout << "\t\tERROR\n";

  cout << "crt          --\n";
  (crt2(2, 5, 3, 13) == 42 ) ? cout << "\t\tokay\n" : cout << "\t\tERROR\n";

  cout << "square root  --\n";
  ((sqrt2(4, 7, 11) == 9) ||
   (sqrt2(4, 7, 11) == 2) ||
   (sqrt2(4, 7, 11) == 75) ||
   (sqrt2(4, 7, 11) == 68)) ? cout << "\t\tokay\n" : cout << "\t\tERROR\n";

  cout << "legendre     --\n";
  (legendre(4, 23) == 1) ? cout << "\t\tokay\n" : cout << "\t\tERROR\n";
  (legendre(5, 7) == -1) ? cout << "\t\tokay\n" : cout << "\t\tERROR\n";

  cout << "Rabin sign   --\n";
  total_usecs = 0;
  for (repetitions=1; repetitions <=REPEAT; repetitions++){
    gen_rabin_key(&rabin_s);
    rabin_p.n = rabin_s.n;
    do {
      temp = generate_random((SIZE_OF_KEYS-1)*2);
    } while (!((legendre(temp, rabin_s.p) == 1) && (legendre(temp, rabin_s.q) == 1)));
    m.msg = temp;
    rabin_sign(rabin_s, &m);
    gettimeofday(&thing, NULL);
    (rabin_check(rabin_p, m)) ? cout << "." : cout << "X";
    gettimeofday(&thing2, NULL);
    secs = thing2.tv_sec - thing.tv_sec;
    usecs = thing2.tv_usec - thing.tv_usec;
    if (usecs < 0) {
      usecs += 1000000;
      secs -= 1;
    }
    total_usecs = total_usecs + secs*1000000 + usecs;
  }
  secs=0; usecs=0;
  total_usecs /= REPEAT;
  while (total_usecs > 1000000) {
    secs += 1;
    total_usecs -= 1000000;
  }
  usecs = total_usecs;
  cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";

  cout << "Mod-Rabin -- valid signatures  --\n";
  total_usecs = 0;
  for (repetitions=1; repetitions <=REPEAT; repetitions++){
    gen_rabin_key(&rabin_s);
    rabin_p.n = rabin_s.n;
    do {
      temp = generate_random((SIZE_OF_KEYS-1)*2);
    } while (!((legendre(temp, rabin_s.p) == 1) && (legendre(temp, rabin_s.q) == 1)));
    m.msg = temp;
    mod_rabin_sign(rabin_s, &m);
    gettimeofday(&thing, NULL);
    (mod_rabin_check(rabin_p, m, NUMBER_TRIES)) ? cout << "." : cout << "X";
    gettimeofday(&thing2, NULL);
    secs = thing2.tv_sec - thing.tv_sec;
    usecs = thing2.tv_usec - thing.tv_usec;
    if (usecs < 0) {
      usecs += 1000000;
      secs -= 1;
    }
    total_usecs = total_usecs + secs*1000000 + usecs;
```

```
}
total_usecs /= REPEAT;
secs=0; usecs=0;
while (total_usecs > 1000000) {
  secs += 1;
  total_usecs -= 1000000;
}
usecs = total_usecs;
cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";

cout << "Mod-Rabin, quick -- valid signatures  --\n";
total_usecs = 0;
for (repetitions=1; repetitions <=REPEAT; repetitions++){
  gen_rabin_key(&rabin_s);
  rabin_p.n = rabin_s.n;
  do {
    temp = generate_random((SIZE_OF_KEYS-1)*2);
  } while (!((legendre(temp, rabin_s.p) == 1) && (legendre(temp, rabin_s.q) == 1)));
  m.msg = temp;
  mod_rabin_sign(rabin_s, &m);
  init_quick_sig(NUMBER_TRIES);
  (check_sig_quick(rabin_p, m, &thing)) ? cout << "." : cout << "X";
  gettimeofday(&thing2, NULL);
  secs = thing2.tv_sec - thing.tv_sec;
  usecs = thing2.tv_usec - thing.tv_usec;
  if (usecs < 0) {
    usecs += 1000000;
    secs -= 1;
  }
  total_usecs = total_usecs + secs*1000000 + usecs;
}
total_usecs /= REPEAT;
secs=0; usecs=0;
while (total_usecs > 1000000) {
  secs += 1;
  total_usecs -= 1000000;
}
usecs = total_usecs;
cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";

cout << "Mod-Rabin -- bogus signatures  --\n";
total_usecs = 0;
for (repetitions=1; repetitions <=REPEAT; repetitions++){
  gen_rabin_key(&rabin_s);
  rabin_p.n = rabin_s.n;
  do {
    temp = generate_random((SIZE_OF_KEYS-1)*2);
  } while (!((legendre(temp, rabin_s.p) == 1) && (legendre(temp, rabin_s.q) == 1)));
  m.msg = temp;
  m.sig = generate_random(SIZE_OF_KEYS-1);
  gettimeofday(&thing, NULL);
  (mod_rabin_check(rabin_p, m, NUMBER_TRIES)) ? cout << "X" : cout << ".";
  gettimeofday(&thing2, NULL);
  secs = thing2.tv_sec - thing.tv_sec;
  usecs = thing2.tv_usec - thing.tv_usec;
  if (usecs < 0) {
    usecs += 1000000;
    secs -= 1;
  }
  total_usecs = total_usecs + secs*1000000 + usecs;
}
total_usecs /= REPEAT;
secs=0; usecs=0;
while (total_usecs > 1000000) {
  secs += 1;
  total_usecs -= 1000000;
}
usecs = total_usecs;
cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";

cout << "Mod-Rabin, quick -- bogus signatures  --\n";
total_usecs = 0;
for (repetitions=1; repetitions <=REPEAT; repetitions++){
  gen_rabin_key(&rabin_s);
  rabin_p.n = rabin_s.n;
  do {
    temp = generate_random((SIZE_OF_KEYS-1)*2);
  } while (!((legendre(temp, rabin_s.p) == 1) && (legendre(temp, rabin_s.q) == 1)));
  m.msg = temp;
  m.sig = generate_random(SIZE_OF_KEYS-1);
  init_quick_sig(NUMBER_TRIES);
  (check_sig_quick(rabin_p, m, &thing)) ? cout << "X" : cout << ".";
```

```
    gettimeofday(&thing2, NULL);
    secs = thing2.tv_sec - thing.tv_sec;
    usecs = thing2.tv_usec - thing.tv_usec;
    if (usecs < 0) {
      usecs += 1000000;
      secs -= 1;
    }
    total_usecs = total_usecs + secs*1000000 + usecs;
  }
  total_usecs /= REPEAT;
  secs=0; usecs=0;
  while (total_usecs > 1000000) {
    secs += 1;
    total_usecs -= 1000000;
  }
  usecs = total_usecs;
  cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";

  cout << "RSA sign     --\n";
  total_usecs = 0;
  for (repetitions=1; repetitions <=REPEAT; repetitions++){
    gen_rsa_key(&rsa_s);
    rsa_p.n = rsa_s.n;
    rsa_p.e = rsa_s.e;
    m.msg = generate_random(SIZE_OF_KEYS - 1);
    rsa_sign(rsa_s, &m);
    gettimeofday(&thing, NULL);
    (rsa_check(rsa_p, m)) ? cout << "." : cout << "X";
    gettimeofday(&thing2, NULL);
    secs = thing2.tv_sec - thing.tv_sec;
    usecs = thing2.tv_usec - thing.tv_usec;
    if (usecs < 0) {
      usecs += 1000000;
      secs -= 1;
    }
    total_usecs = total_usecs + secs*1000000 + usecs;
  }
  total_usecs /= REPEAT;
  secs=0; usecs=0;
  while (total_usecs > 1000000) {
    secs += 1;
    total_usecs -= 1000000;
  }
  usecs = total_usecs;
  cout << "  average time =  " << secs << " secs : " << usecs << " usecs\n";
}
```

utils.h

```
#include <Integer.h>

#define PRIMES_IN_FILE 50 // number of primes in a pregenerated file
#define LARGEST_PRIME_DIGITS 1024 // largest size expecting to find
                                  // in prime_file.xk
#define TIMES_TO_REPEAT 5 // times to repeat Miller-Rabin

// prototypes of stuff in utils.C
extern void          init            ();
extern Integer       exp_mod         (Integer, Integer, Integer);
extern long          exp_mod         (long, long, long);
extern Integer       generate_prime  (int);
extern Integer       generate_random (int);
extern Integer       crt             (Integer [], Integer [], Integer);
extern Integer       crt2            (Integer, Integer, Integer, Integer);
extern Integer       sqrt            (Integer, Integer[], int);
extern Integer       sqrt2           (Integer, Integer, Integer);
extern Integer       inverse         (Integer, Integer);
extern int           legendre        (Integer, Integer);
extern int           legendre        (long, long);
```

utils.C

```
extern "C" {
  #include <stdio.h>
  #include <time.h>
  #include <sys/time.h>
  #include <sys/param.h>
  #include <stdlib.h>
```

```
}
#include <Integer.h>
#include "utils.h"

int small_primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
                       43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
                       101, 103, 107, 109, 113, 127, 131, 137, 139,
                       149, 151, 157, 163, 167, 173, 179, 181, 191,
                       193, 197, 199, 211, 223, 227, 229, 233, 239,
                       241, 251, 257, 263, 269, 271, 277, 281, 283,
                       293, 307, 311, 313, 317, 331, 337, 347, 349,
                       353, 359, 367, 373, 379, 383, 389, 397, 401,
                       409, 419, 421, 431, 433, 439, 443, 449, 457,
                       461, 463, 467, 479, 487, 491, 499, 503, 509,
                       521, 523, 541, 547, 557, 563, 569, 571, 577,
                       587, 593, 599, 601, 607, 613, 617, 619, 631,
                       641, 643, 647, 653, 659, 661, 673, 677, 683,
                       691, 701, 709, 719, 727, 733, 739, 743, 751,
                       757, 761, 769, 773, 787, 797, 809, 811, 821,
                       823, 827, 829, 839, 853, 857, 859, 863, 877,
                       881, 883, 887, 907, 911, 919, 929, 937, 941,
                       947, 953, 967, 971, 977, 983, 991, 997, 1009,
                       1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051,
                       1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103,
                       1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171,
                       1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229,
                       1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289,
                       1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327,
                       1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427,
                       1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471,
                       1481, 1483, 1487, 1489, 1493, 1499, 1511, 1523,
                       1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579,
                       1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621,
                       1627, 1637, 1657, 1663, 1667, 1669, 1693, 1697,
                       1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753,
                       1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823,
                       1831, 1847, 1861, 1867, 1871, 1873, 1877, 1879,
                       1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951,
                       1973, 1979, 1987, 1993, 1997, 1999, 0 };

// Prototypes of functions that should only be accesses from here
Integer       sqrt_p          (Integer, Integer);
int           rabin_miller    (Integer, int);
void          e_euclid        (Integer, Integer, Integer *, Integer *, Integer *);

void init()
/* initializer function. This should be called before anything else
 * to seed the random number generator and do whatever other necessary
 * things need to be done.
 */
{
  struct timeval foo;

  gettimeofday(&foo, NULL);
  srandom(foo.tv_sec + foo.tv_usec);
}

Integer exp_mod(Integer a, Integer b, Integer m)
/* This function does a^b (mod m), in hopefully a slightly fast
 * way through repeated exponentiation and modulo in appropriate
 * times (using x^2^2 etc..).
 */
{
  Integer s = 1;
  while (b != 0) {
    if (odd(b))
      s = (s*a) % m;
    b >>= 1;
    a = (a*a) % m;
  }
  return (s);
}

long exp_mod(long a, long b, long m)
/* This function does a^b (mod m), in hopefully a slightly fast
 * way through repeated exponentiation and modulo in appropriate
 * times (using x^2^2 etc..).
 * it won't work for anything greater than 15 bit numbers....
 */
{
  long s = 1;
  while (b != 0) {
```

```
    if (odd(b))
      s = (s*a) % m;
    b >>= 1;
    a = (a*a) % m;
  }
  return (s);
}

Integer generate_random(int size)
/* This function generates a random number of size bits
 *  This is NOT a good random number generator right
 *  now.. fixing it will be nice, but unnecessary for
 *  testing purposes.
 */
{
  Integer value, mask, temp=size;
  value = 0;

  while (temp > 0){
    temp -= 32;
    value <<= 32;
    value += random();
  }
  mask = (1<<(Integer)size) - 1;     // set all the bits up to size'th to 1
  value &= mask;                     // zero out all the larger bits

  return value;
}

Integer generate_prime(int size)
/* this function generates a prime of the size passed in.
 * It relies on a random number generator, and uses the following
 * scheme:
 *  1) generate a size-bit number p
 *  2) set the size'th bit and bit 0 to 1 (odd and of set size).
 *  3) check to ensure it doesn't devide by small primes
 *  4) do a Rabin-Miller test five times.
 */
{
  int good = 0;
  Integer p, a, b, m;
  FILE *prime_file;
  char name[MAXPATHLEN];
  char value[LARGEST_PRIME_DIGITS];  // okay..this shouldn't be static but that's fast.

  sprintf(name, "primes.%d", size);
  if ((prime_file = fopen(name, "r")) != NULL) { // get the prime from the file
    good=(random() % PRIMES_IN_FILE);
    while (good--){
      fgets(value, LARGEST_PRIME_DIGITS-1, prime_file);
    }
    fgets(value, LARGEST_PRIME_DIGITS-1, prime_file);
    p = atoI(value);
    fclose(prime_file);
  }
  else{
    while (!good) {
    try_again:
      good = 1;

      p = generate_random(size);
      (setbit)(p, (size-1));      // make sure p is fully size bits
      (setbit)(p, 0);             // make it odd

      // p should now be a large random odd number of size bits.
      // test that it doesn't devide by small primes

      /*
         for (int index = 0; small_primes[index]; index++) {
         if ((p % small_primes[index]) == 0)
         goto try_again;
         }
         */
      if ((p % 3) == 0)
        goto try_again;
      if ((p % 5) == 0)
        goto try_again;
      if ((p % 7) == 0)
        goto try_again;
      if ((p % 11) == 0)
        goto try_again;
      if ((p % 13) == 0)
```

```
        goto try_again;
      if ((p % 17) == 0)
        goto try_again;
      if ((p % 19) == 0)
        goto try_again;
      if ((p % 23) == 0)
        goto try_again;
      if ((p % 29) == 0)
        goto try_again;
      if ((p % 31) == 0)
        goto try_again;
      if ((p % 37) == 0)
        goto try_again;

      // okay.. so it doesn't have a small factor, now do a probabilistic
      // test, and cross your fingers
      // Miller-Rabin test:

      if (!(rabin_miller(p, TIMES_TO_REPEAT)))
        goto try_again;
    }
  }
  return(p);
}

int rabin_miller(Integer p, int repeat)
/* Test whether p is indeed a prime using the rabin_miller scheme, with
 * one pass. It is reccomended you call this function at least 5 times.
 */
{
  Integer m, a, z;
  int j, b = 0;

  //b is the number of times p-1 devides into 2
  m = p-1;
  while (testbit(m, b) == 0){
    b++;
  }
  m /= (pow(2, b));

  while (repeat--){
    a = generate_random(32);  // a should be smaller than p...
    j = 0;
    z = exp_mod(a,m,p);
    if (!((z==1) || (z == p-1))){
      while(1){  // repeat until convinced of one or the other
        if ((j > 0) && (z == 1))
          return(0);  // not prime
        j++;
        if ((j < b) && (z != p-1))
          z = exp_mod(z, 2, p);
        else
          if ((j == b) && (z != p-1))
            return(0);  // not prime
          else
            return(1);  // prime
      }
    }
  }
  return(1);
}

Integer crt(Integer m[], Integer c[], int num_elements)
/* This function computes the chinese remainder theorem result
 *  for    result = c[0] % m[0]
 *         result = c[1] % m[1]
 *  etc.
 * the arrays passed in are the values, and num_elements is the
 *  number of elements in the array.
 */
{
  int i;
  Integer modulus, n, a;

  modulus = 1;
  for (i = 0; i < num_elements; i++)
    modulus *= m[i];
  n = 0;
  for (i=0; i < num_elements; i++){
    a = modulus/m[i];
    n += c[i] * a * (inverse(a,m[i]));
    n %= modulus;
```

```
  }
  return n;
}

Integer crt2(Integer value1, Integer p, Integer value2, Integer q)
/* convenience routine to access crt for val1 % p && val2 % q
 *  calls crt, and returns the result
 */
{
  Integer mods[2], coef[2];

  mods[0] = p;
  mods[1] = q;
  coef[0] = value1;
  coef[1] = value2;

  return(crt(mods, coef, 2));
}

Integer sqrt(Integer value, Integer modulus[], int k)
/* This function calculates the square root of a value in a field
 * where m[0]*m[1]*...*m[k] = n
 * for now, there is another caveat: m[i] = 3 mod 4. The alternative
 *   is much harder for me to code :-)
 */
{
  Integer roots[k];
  int t = k;

  while (t--)
    if ((modulus[t] % 4) != 3){
      cerr<< "sqrt: not 3 mod 4 prime!";
      return(0); // error
    }

  if (k>1)
    {
      for (t=0; t<k; t++)
        roots[t] = sqrt_p(value, modulus[t]);
      return(crt(modulus, roots, k));
    }
  else
    return(sqrt_p(value, modulus[0]));
}

Integer sqrt2(Integer val, Integer p, Integer q)
/* Same as above for m = p*q
 */
{
  Integer roots[2];
 if (((p%4) != 3) || ((q%4) !=3)) {
    cerr<< "sqrt2: not 3 mod 4 prime!";
    return(0); // error
  }
  return(crt2(sqrt_p(val, p), p, sqrt_p(val, q), q));
}

Integer sqrt_p(Integer value, Integer prime)
/* This function calculates sqrt(x) mod p. It should only be called
 *  from the sqrt function above, since we assume that p = 3 mod 4
 */
{
  Integer m;

  return(exp_mod(value, ((((prime-1)/2) +1)/2), prime));
}

Integer inverse(Integer a, Integer p)
/* Computes x s.t.  x*a = 1 (mod p) and returns that as the result.
 *  assumes that p is prime. uses Extended Euclid.
 */
{
  Integer x, d, y;
  e_euclid(a, p, &d, &x, &y);
  if (d != 1) {
    cerr << "inverse: gcd is NOT 1\n";
    return(0);
  }
  if (x<0)
    return(p + x);
  else
    return(x);
```

```
}

void e_euclid(Integer a, Integer b, Integer *d, Integer *x, Integer *y)
/* This computes d=ax+by using extended euclid gcd algorithm. from CLR
 */
{
  Integer t, u, v;

  if (b==0){
    *d = a;
    *x = 1;
    *y = 0;
    return;
  }
  e_euclid(b, a%b, &t, &u, &v);
  *d = t;
  *x = v;
  *y = u - ((a/b) * v);
  return;
}

int legendre(Integer number, Integer prime)
/* This computes the legendre symbol of a given number, using the
 * n ^ ((p-1)/2) method.
 */
{
  Integer a;
  a = exp_mod(number, ((prime-1)/2), prime);
  if (a==1)
    return 1;
  if ((a==0) || (a==prime))
    return 0;
  if ((a==-1) || (a== prime-1));
    return -1;
}

int legendre(long number, long prime)
/* This computes the legendre symbol of a given number, using the
 * n ^ ((p-1)/2) method.
 */
{
  long a;
  a = exp_mod(number, ((prime-1)/2), prime);
  if (a==1)
    return 1;
  if ((a==0) || (a==prime))
    return 0;
  if ((a==-1) || (a== prime-1));
    return -1;
}
```