

MiniPASS: Authentication and Digital Signatures in a Constrained Environment

Jeffrey Hoffstein, Joseph H. Silverman

NTRU Cryptosystems, Inc., 3 Leicester Way, Pawtucket, RI 02860 USA
jhoff@ntru.com, jhs@ntru.com <www.ntru.com>

Abstract. We describe an implementation of the PASS polynomial authentication and signature scheme [5, 6] that is suitable for use in highly constrained environments such as SmartCards and Wireless Applications. The algorithm underlying the PASS scheme, as described in [5, 6], already features high speed and a small footprint, and these are further enhanced by transferring computational overhead to the Server to the extent possible. We also describe timing and footprint results from a prototype implementation.

Introduction

Secure public key authentication and digital signatures are increasingly important for electronic communications and commerce, and they are required not only on high powered desktop computers, but also on SmartCards and wireless devices with severely constrained memory and processing capabilities. An authentication/digital signature scheme called PASS (Polynomial Authentication and Signature Scheme) was introduced in [5], and a slightly modified version with even better operating characteristics was described in [6]. It was asserted in [5, 6] that PASS is ideal for constrained environments due to its high speed and small footprint. In this article we substantiate those claims by giving a detailed description of how to implement PASS on a small memory/low speed device such as a SmartCard. We also give the results of experiments using a preliminary implementation of these ideas.

The importance of public key authentication and digital signatures is amply demonstrated by the large literature devoted to both theoretical and practical aspects of the problem, see for example [2, 3, 8, 9, 11, 13, 14, 16]. The widespread need for such applications makes the introduction of new schemes of interest to both the academic and financial communities, especially schemes which are based on much studied hard mathematical problems and which offer significant practical advantages in terms of speed and key size over existing methods.

1 A Brief Description of PASS

The PASS Polynomial Authentication and Signature Scheme is based on the hard mathematical problem of finding a binary polynomial $f(X)$ that takes

on prescribed values $f(\alpha) \bmod q$ are at a given collection of numbers $\alpha = \alpha_1, \alpha_2, \dots, \alpha_n$. (This problem is equivalent to solving the closest vector problem in a certain lattice, see [5, 6] for details.) Briefly, the Prover publishes the set of values $f(\alpha_i)$ as her public key, and she proves her identity by demonstrating that she possesses a binary polynomial taking those values.

One version of PASS was presented at CryptTEC '99 [5] and a modification based on the same principles, but with better operating characteristics, is described in [6]. Since our goal in this paper is to fit a fast and secure authentication scheme into a highly constrained environment, we will use the version of PASS from [6], but virtually all of our remarks apply also to the original version in [5].

In this section we will briefly review how the PASS scheme works. Further information and a detailed security analysis may be found in the cited papers.

A PASS scheme depends on the choice of a prime number q , and we set $N = q - 1$. A typical choice yielding a security level approximately equivalent to an RSA 1024 bit key (see [5, 6] for a detailed security analysis) is

$$q = 769 \quad \text{and} \quad N = 768. \quad (1)$$

For higher security, one might take $q = 1053$ and $N = 1052$. For the remainder of this article, unless we specify otherwise, all computations with numbers are performed modulo q .

The basic objects used by PASS are polynomials of degree $N - 1$,

$$f(X) = a_0 + a_1X + a_2X^2 + \dots + a_{N-1}X^{N-1},$$

taken with coefficients modulo q . Multiplication is accomplished using the rule $X^N = 1$, which leads to the multiplication formula

$$\left(\sum_{i=0}^{N-1} a_i X^i \right) \left(\sum_{i=0}^{N-1} b_i X^i \right) = \sum_{k=0}^{N-1} \left(\sum_{\substack{i+j \equiv k \pmod{N} \\ 0 \leq i, j < N}} a_i b_j \right) X^k. \quad (2)$$

Another way to view this multiplication is to write the coefficients of a polynomial as a vector

$$[a_0, a_1, \dots, a_{N-1}],$$

and then the product of two vectors is the usual convolution product.

The other public parameter for PASS is a set

$$S = \{\alpha_1, \alpha_2, \dots, \alpha_{N/2}\}$$

of distinct nonzero numbers modulo q with the property that if $\alpha \in S$, then also $\alpha^{-1} \in S$. For concreteness, we will fix a generator w modulo q (i.e., w is a primitive root modulo q) and take

$$S = \{w^i : N/4 \leq i \leq 3N/4\}. \quad (3)$$

In particular, there is no need to actually store the set S .

If $f(X)$ is a polynomial of degree $N-1$ with mod q coefficients as above, then its *Discrete Fourier Transform* (DFT) is a polynomial $\hat{f}(X)$ whose coefficients are the values of f . More precisely, we fix a generator w modulo q , and then

$$\hat{f}(X) = \sum_{j=0}^{N-1} f(w^j)X^j,$$

where remember that all numbers are computed modulo q . A well-known formula says that the coefficients of the original polynomial $f(X) = \sum a_i X^i$ can be recovered from the values of f via the equation

$$a_i = \frac{1}{N} \hat{f}(w^{-i}) = \frac{1}{N} \sum_{j=0}^{N-1} f(w^j)w^{-ij}. \quad (4)$$

(Here w^{-1} is the inverse of w modulo q , and w^{-i} is the i^{th} power of w^{-1} .)

Now suppose that you are only given some of the values of f , for example the set of values

$$f(S) = \{f(\alpha) : \alpha \in S\}.$$

There are a large number of polynomials which take these prescribed values (precisely, there will be $q^{N/2}$ of them). However, the PASS polynomial forming the private key will have the additional property that it is a *binary polynomial*, that is, all of its coefficients are 0 or 1. It is then a difficult problem to find the target binary polynomial $f(X)$ among the $q^{N/2}$ polynomials taking the correct values.

Remark 1. The security of PASS is based on the fact that it is difficult to simultaneously control both the values and the coefficients of a polynomial over a finite field. As indicated above, the values and the coefficients of a polynomial are (discrete) Fourier transforms of one another. Thus underlying PASS is the mathematical principle that it is difficult to simultaneously control the values of a function and the values of its Fourier transform. This principle is the discrete analogue (for finite fields) of the Heisenberg Uncertainty Principle. As described in [5, 6, 12], it can be solved using lattice reduction methods (just as RSA can be broken using the number field sieve), but if N is sufficiently large, then the underlying lattice problem is too difficult to solve using current techniques.

Outline of the PASS Authentication and Signature Scheme

Public Parameters All users agree on a prime number q and a set of distinct numbers $S = \{\alpha_1, \dots, \alpha_{N/2}\}$ modulo q , and they let $N = q - 1$. All polynomials are of degree $N - 1$, polynomial multiplication uses the convolution rule (2) given above, and all computations (except for verification step A) are performed modulo q . Appropriate quantities A_h, B_h are also chosen to be used in the verification process.

Key Creation The Prover selects a binary polynomial $f(X)$. This polynomial is her private key. She publishes the set of values $f(S) = \{f(\alpha) : \alpha \in S\}$. This set of values is her public key.

Commitment In the commitment step, the Prover selects another binary polynomial $g_1(X) \in R_q$. She computes and sends to the Verifier the set of values $g_1(S)$.

Challenge In the challenge step, the Verifier selects two extremely small polynomials $c_1(X)$ and $c_2(X)$ (say with between two and eight nonzero coefficients) and sends them to the Prover. For security reasons, it is also important that $c_1(X)$ have no nonzero roots modulo q for values of X not in S . (There is at least a 50% chance that this will be true for a randomly chosen c_1 .)

Response In the response step, the Prover selects a third binary polynomial $g_2(X)$. She computes and sends to the Verifier the polynomial

$$h(X) = (f(X) + c_1(X)g_1(X) + c_2(X)g_2(X))g_2(X). \quad (5)$$

Verification The Verifier performs the following two steps to verify the Prover's identity:

(A) The Verifier checks that the polynomial $h(X)$ is moderately small by writing it as $h(X) = \sum a_i X^i$ and verifying the bound

$$\sum_{i=0}^{N-1} (a_i - A_h)^2 < B_h,$$

where A_h and B_h are public quantities. Note that this computation is not done modulo q , but is simply a sum of integers.

(B) For each $\alpha \in S$, the Verifier computes the quantity

$$(f(\alpha) + c_1(\alpha)g_1(\alpha))^2 + 4c_2(\alpha)h(\alpha) \pmod{q} \quad (6)$$

and checks that it is a square modulo q .

If the polynomial h passes tests (A) and (B), then the Verifier accepts the Prover's identity.

Why It Works First we note that the definition (5) says that the polynomial $h(X)$ is a simple combination of the polynomials f, g_1, g_2, c_1, c_2 , all of which are binary, so it is clear that the coefficients of $h(X)$ will also be moderately small. It is a simple matter to take A_h to be the average expected value of the coefficients of h and to experimentally find a bound B_h so that if h has the correct form (5), then it will almost certainly pass verification step A. Next we observe that the verifier is able to compute the quantity (6), since he knows the polynomials $h(X), c_1(X), c_2(X)$ and he knows the values of $f(X)$ and $g_1(X)$ for all $\alpha \in S$. To see why (6) is a square, we use the definition (5) of $h(X)$ to compute

$$\begin{aligned} (f + c_1g_1)^2 + 4c_2h &= (f + c_1g_1)^2 + 4c_2((f + c_1g_1 + c_2g_2)g_2) \\ &= f^2 + 2c_1g_1 + c_1^2g_1^2 + 4c_2g_2f + 4c_1c_2g_1g_2 + 4c_2^2g_2^2 \\ &= (f + c_1g_1 + 2c_2g_2)^2. \end{aligned} \quad (7)$$

Thus $(f(X) + c_1(X)g_1(X))^2 + 4c_2(X)h(X)$ is actually the square of a polynomial, so it certainly gives a square modulo q when evaluated at any α .

Remark 2.

- The best way to create the challenge polynomials c_1 and c_2 is for the Verifier to choose a random string (of 80 to 160 bits) and send it to the Prover. They then both apply a common hash function to the random string in order to create c_1 and c_2 . This has the advantage of cutting down the number of bits transmitted, as well as making it more difficult for the Verifier to mount any sort of attack based on choosing c_1 and c_2 to have a particular form.
- We have presented PASS as an authentication scheme, but any authentication scheme that includes a challenge step can be combined with a hash function to create a signature scheme. Thus if a digital document D is to be signed, it is sent through a standard hash function to obtain a small bit string (say 80 to 160 bits), and this bit string is hashed to create the challenge polynomials c_1, c_2 . The Signer then publishes the values $g_1(S)$, $h(X)$, and D . (We assume, of course, that the Signer’s public key $f(S)$ is already in the public domain.) Anyone wishing to verify the signature can use D and the hash functions to recreate c_1 and c_2 , and then he has enough information to perform the verification step described above.
- The computation of the sets of values $f(S)$, $g_1(S)$, and $h(S)$ required during the PASS process can be performed extremely rapidly. Even a direct computation takes only $O(N^2)$ steps, but it is even more efficient to use FFT, especially if N is highly divisible by 2 (as are the suggested values $N = 768$, $N = 928$, and $N = 1052$), in which case the computation is reduced to $O(N \log N)$ steps. Note that the field \mathbb{F}_q contains a primitive N^{th} -root of unity, so in this setting one can compute Fast Fourier Transforms using only integer arithmetic; there is no need to use complex numbers or floating point numbers.
- In verification step (B), the Verifier needs to check if certain quantities are squares. This can be done rapidly using either quadratic reciprocity or the powering map, but if a little extra storage is available, it’s even quicker to precompute a table of squares.

2 MiniPASS

Our goal is to fit PASS into a minimal amount of space while still retaining desirable operating characteristics. We begin by analyzing each step of the PASS algorithm to see how much computation needs to be done. In particular, we will assume that a SmartCard with constrained operating resources is communicating with a Server that has access to a more robust operating environment. Thus we will analyze PASS twice, first with the SmartCard as the Prover, and second with the SmartCard as the Verifier. In both cases we will shift as much of the computation as possible onto the Server. We will assume that the SmartCard already include a (pseudo)random number generator and a hash function.

2.1 The SmartCard As Prover

We will assume that the SmartCard's public key $f(S)$ is publicly available, and that her private key $f(X)$ is stored in ROM. Note that since the private key $f(X)$ is a binary polynomial, it requires only N bits to store.

The SmartCard/Prover begins with the Commitment step. She chooses a random binary polynomial $g(X)$. She needs to compute and send to the Server the values $g(\alpha)$ for every $\alpha \in S$. If $g(X) = \sum b_i X^i$, she can compute these values one at a time via the formula

$$g(\alpha) = (\cdots((a_{N-1} * \alpha + a_{N-2}) * \alpha + a_{N-3}) * \alpha + \cdots + a_1) * \alpha + a_0. \quad (8)$$

This method requires N multiplications modulo q and N additions. There are much faster ways to compute the $g(\alpha)$ values (see remark 5 below), but they require somewhat more storage, which is what we are trying to minimize. Note that the SmartCard/Prover does not need to store the values, so she can simply compute one $g(\alpha)$, send that value to the Server/Verifier, and then go on to the next value of α .

The Server/Verifier then selects challenge polynomials c_1 and c_2 , or more likely, selects a bit string that is hashed to form c_1 and c_2 . See Remark 3 below for further details on the selection of c_1 and c_2 .

In the response step, the SmartCard/Prover is supposed to select another binary polynomial $g_2(X)$ and send the quantity

$$h = (f + c_1 g_1 + c_2 g_2) g_2 = f g_2 + c_1 g_1 g_2 + c_2 g_2^2$$

to the Server/Verifier. However, it is probably more efficient for the SmartCard/Prover to compute and transmit all of the values of this polynomial h , and then the Server/Verifier can reconstruct h itself using the inversion formula (4). As in the commitment step, the SmartCard/Prover only needs to compute one value of $h(\alpha)$ at a time. Further, the challenge polynomials c_1 and c_2 are extremely sparse, so evaluating them can be done very rapidly. Thus the time consuming part of the response step is computation of the values $f(\alpha)$, $g_1(\alpha)$, $g_2(\alpha)$ for all $0 \leq \alpha < q$, but even this is not a tremendously onerous task.

The SmartCard/Prover has now fulfilled her tasks, and it remains for the Server/Verifier to perform the final verification step.

2.2 The SmartCard As Verifier

We will assume that the SmartCard/Verifier contains the Server/Prover's public key $f(S)$ in ROM. In principle, this requires $\frac{N}{2} \log_2(q)$ bits; but in practice for (say) $q = 769$, each of the $N/2$ numbers modulo q would be stored in 16 bits, so the public key requires N bytes of storage.

The first thing that the SmartCard/Verifier does is receive from the Server/Prover the set of values $g_1(\alpha)$ for $\alpha \in S$, and it appears that the SmartCard/Verifier needs to store all of those values for later use. This seems necessary

because in the final verification step, the SmartCard/Verifier is asked to check that the quantity

$$(f(\alpha) + c_1(\alpha)g_1(\alpha))^2 + 4c_2(\alpha)h(\alpha) \pmod{q}$$

is a square modulo q for every $\alpha \in S$. However, suppose that instead the SmartCard/Verifier only checks the condition (2.2) for a random selection of $\alpha \in S$. More precisely, suppose that she checks (say) 60 values of α and that all of them pass the test (2.2). The probability of this happening at random is 2^{-60} , so she can be fairly confident that in fact every $\alpha \in S$ will pass the test (2.2). This simple observation will greatly increase operating speed while decreasing memory requirements.

This means that prior to the commitment step, the SmartCard/Verifier randomly selects a set of 60 numbers

$$T = \{\alpha_1, \alpha_2, \dots, \alpha_{60}\}$$

in S . (For added efficiency, but slightly reduced security, she could instead select 40 numbers.) Then, during the commitment step, the SmartCard/Verifier will receive from the Server/Prover the values $g_1(\alpha)$ for every $\alpha \in S$, but she will only store the 60 values of $g_1(\alpha)$ for $\alpha \in T$.

For the challenge step, the SmartCard/Verifier creates the polynomials $c_1(X)$ and $c_2(X)$ and sends them to the Server/Prover. More precisely, she chooses a random string, and c_1 and c_2 are created using a hash function, see Remark 3 below for details.

The Server/Prover's response is to create a certain polynomial $h(X)$ and send $h(X)$ to the SmartCard/Verifier. If $h(X)$ is the polynomial

$$h(X) = a_0 + a_1X + a_2X^2 + \dots + a_{N-1}X^{N-1},$$

we will require the Server/Prover to transmit $h(X)$ to the SmartCard/Verifier as the list of coefficients $a_{N-1}, a_{N-2}, \dots, a_1, a_0$. As the SmartCard/Verifier receives each coefficient, she does two things. First, she keeps a running total of the quantities

$$(a_i - A_h)^2, \quad i = 0, 1, 2, \dots, N-1. \quad (9)$$

Note that these numbers are not reduced modulo q , so the sum should be stored as a 32 bit number. Second, she computes the values of $h(\alpha)$ modulo q one coefficient at a time, but only for the 60 values of α in T . Note that she does not need to store the coefficients of h , so the only storage requirements are the running total (9) and the 60 values of h , for a total of $4 + 60 \cdot 2 = 124$ bytes.

After receiving and storing this information, it remains to complete the verification process. If the running total (9) is larger than B_h , then the Server/Prover's identity is rejected. Otherwise, the SmartCard/Verifier computes the quantity

$$(f(\alpha) + c_1(\alpha)g_1(\alpha))^2 + 4h(\alpha)c_2(\alpha) \quad (10)$$

for each $\alpha \in T$ and checks if it is a square modulo q . Note that the SmartCard/Verifier stored the values of $g_1(\alpha)$ for $\alpha \in T$ during the commitment step, she stored the values of $h(\alpha)$ for $\alpha \in T$ during the response step, and she knows f, c_1, c_2 , so she can compute their values for any α . It is thus easy (and fast) for her to compute the 60 values of (10) for the numbers $\alpha \in T$.

There remains the question of how she verifies that they are squares modulo q . The fastest method is to store a table of values, or more efficiently, store a bit string of length $q - 1$ so that the i^{th} bit equals 1 if i is a square modulo q . For $q = 769$, this requires an additional 96 bytes of ROM. An alternative method for checking if a number n is a square modulo q is to compute

$$n^{(q-1)/2} \pmod{q}.$$

This value will be 1 if n is a square, and -1 if it is not a square. This powering operation is fast, and will probably already be used for computing the values of the sparse polynomials $c_1(X)$ and $c_2(X)$, so it will not require additional routines. (It is, of course, also possible to use quadratic reciprocity for this step.)

We stress again that since the SmartCard/Verifier is only checking 60 values, the time required to perform a verification is extremely small. Based on the experiments described in Section 3, the SmartCard does verifications 25 to 30 times faster than proving identity.

2.3 Additional Implementation Considerations

We briefly mention additional items to consider during implementation.

Remark 3. In order to avoid possible attacks, the challenge polynomials c_1 and c_2 should be generated as follows. The Verifier selects a random 80 bit string B . A hash function is evaluated at B , and the result is fed into a simple function that generates two very sparse binary polynomials $c_1(X)$ and $c_2(X)$. For $N = 768$, it suffices to have a total of eight nonzero coefficients, and for ease of implementation, we will assume that $c_1(X)$ has two nonzero coefficients and that $c_2(X)$ has six nonzero coefficients. Thus $c_1(X)$ looks like

$$c_1(X) = X^{n_1} + X^{n_2}, \tag{11}$$

and $c_2(X)$ looks similar, but with six terms. Note that c_1 and c_2 should be stored as a list of exponents (e.g., $c_1 = (n_1, n_2)$), so they require only 16 bytes of storage.

For security reasons described in [6], it is important that $c_1(X)$ have no roots α modulo q with $\alpha \notin S$. An easy way to guarantee this is to take c_1 as above (11) with the condition that the exponents satisfy $\gcd(N, n_1 - n_2) = 1$. Further, for $N = 768$, this gcd condition is equivalent to

$$n_1 - n_2 \equiv \pm 1 \pmod{6}, \tag{12}$$

so it isn't even necessary to compute a gcd.

Thus a simple protocol for choosing c_1 is to use the hash function as above to produce a possible candidate (11) for c_1 . If it satisfies the security condition (12), stop, otherwise increment n_1 until condition (12) is satisfied. This will take at most 3 iterations.

Remark 4. In order to save space, binary polynomials should be stored as 1 bit per coefficient. Evaluation of binary polynomials via the formula (8) is then moderately inefficient, since individual bits need to be pulled out one at a time. One way to speed up this process is to precompute a small table of (say) 16 values. Thus to compute $f(\alpha)$, first make a table of values:

Bits	Value	Bits	Value	Bits	Value	Bits	Value
0000	0	0100	α^2	1000	α^3	1100	$\alpha^3 + \alpha^2$
0001	1	0101	$\alpha^2 + 1$	1001	$\alpha^3 + 1$	1101	$\alpha^3 + \alpha^2 + \alpha$
0010	α	0110	$\alpha^2 + \alpha$	1010	$\alpha^3 + \alpha$	1110	$\alpha^3 + \alpha^2 + \alpha$
0011	$\alpha + 1$	0111	$\alpha^2 + \alpha + 1$	1011	$\alpha^3 + \alpha + 1$	1111	$\alpha^3 + \alpha^2 + \alpha + 1$

Then read the coefficient bits of $f(X)$ off four bits at a time and use the table to compute a partial value. We illustrate with a polynomial of low degree. If $f(X)$ is the polynomial

$$f(X) = x^{15} + x^{14} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^5 + x^4 + x^2 + 1,$$

then we can evaluate $f(\alpha)$ as

$$([\alpha^3 + \alpha^2 + 1] * \alpha^4 + [\alpha^3 + \alpha + 1]) * \alpha^4 + [\alpha^3 + \alpha + 1] * \alpha^4 + [\alpha^2 + 1].$$

The quantities in square brackets (and the precomputed value of α^4) can be read from the table, significantly increasing efficiency. Since the table only takes 32 bytes, the space is negligible. If additional space is available, one could use 512 bytes to make a table of 256 values; but beyond that size it would make more sense to use Fast Fourier Transforms, which give an even greater speedup.

Remark 5. As indicated in the previous remark, there are various ways to compute the values $f(\alpha)$ of a polynomials that trade space for time. In our situation, since N is divisible by a large power of 2 and since a generator modulo q is an N^{th} root of unity, the fastest way to compute the full set of values $\{f(\alpha) : 1 \leq \alpha < q\}$ is using Fast Fourier Transforms (FFT). This is probably not a good method for use by the SmartCard, since it requires more storage; but the Server will certainly want to use FFT. An FFT polynomial evaluation routine for use by PASS easily fits into 10K (of which only about 4K need be RAM), and at the cost of some efficiency, can be made to fit into 5 or 6K.

3 Sample Implementation of MiniPASS

In this section we describe the results of implementing MiniPASS on a desktop computer using C. We implemented the routines to be used by the SmartCard.

We did not implement the Server routines, which would be considerably faster, but would also require more memory.

For ease of implementation, we used the standard C utility function `rand()` to generate random numbers. This is not cryptographically secure. In practice the Smartcard would probably have its own (pseudo)random number generator and hash function.

Table 1 gives the operating characteristics of our implementation of MiniPASS. We make the following remarks concerning the information in Table 1.

Remark 6.

- ROM includes storage for the SmartCard private key (96 bytes) and for the Server public key (768 bytes).
- The SmartCard never simultaneously acts as Prover and Verifier, so it suffices to have 564 bytes of RAM. At the cost of only checking 40 values (with somewhat reduced security), this may be reduced to 404 bytes.
- The MacOS figures were obtained on a Macintosh G3 300 MHz running MacOS 8.5 and compiled with Metroworks Codewarrior. The Linux figures were obtained on a Celeron 400 MHz running RedHat Linux 6.0 and compiled with egcs.
- We used a table of length 16 as described in Remark 4 to speed evaluation of polynomials. The requisite 32 bytes of RAM is included in the table.

	Card/Prover	Card/Verifier
RAM	350 bytes	564 bytes
ROM (MacOS)	3076 bytes	
ROM (Linux)	3088 bytes	
Time (MacOS)	60.5 ms	2.6 ms
Time (Linux)	71.3 ms	2.3 ms

Table 1. MiniPASS Operating Characteristics

The timing estimates in Table 1 are for computations only. They do not include time for communication between the Smartcard and the Server. The amount of data that needs to be exchanged is listed in Table 2. (We have listed the Challenge as the 20 bytes needed to send the actual challenge polynomials c_1 and c_2 , but in practice the challenge would consist of 80 bits that is hashed to produce the challenge polynomials.)

Remark 7. It would be relatively easy to pack the transmitted material more efficiently and save approximately 37%. This is because the SmartCard and Server are exchanging lists of numbers, with each number lying between 0 and 768. For simplicity, we have assumed that these numbers are stored and transmitted as 16 bit numbers, but they will actually each fit into 10 bits.

Commitment	768 bytes
Challenge	20 bytes
Response	1536 bytes
Total	2324 bytes

Table 2. MiniPASS Communication Requirements

References

1. M. Ajtai, C. Dwork, *A public-key cryptosystem with worst case/average case equivalence*, in Proc. 29th ACM Symposium on Theory of Computing, 1997, 284–293.
2. E.F. Brickell and K.S. McCurley. *Interactive Identification and Digital Signatures*, AT&T Technical Journal, November/December, 1991, 73–86.
3. L.C. Guillou and J.-J. Quisquater. *A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory*, Advances in Cryptology—Eurocrypt '88, Lecture Notes in Computer Science 330 (C.G. Günther, ed.), Springer-Verlag, 1988, 123–128.
4. J. Hoffstein, J. Pipher, J.H. Silverman, *NTRU: A new high speed public key cryptosystem*, in Algorithmic Number Theory (ANTS III), Portland, OR, June 1998, Lecture Notes in Computer Science 1423 (J.P. Buhler, ed.), Springer-Verlag, Berlin, 1998, 267–288.
5. J. Hoffstein, D. Lieman, J.H. Silverman, *Polynomial Rings and Efficient Public Key Authentication*, in Proceeding of the International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99), Hong Kong, (M. Blum and C.H. Lee, eds.), City University of Hong Kong Press.
6. J. Hoffstein, J.H. Silverman, *Polynomial Rings and Efficient Public Key Authentication II*, submitted for publication.
7. O. Goldreich, S. Goldwasser, S. Halevi, *Public-key cryptography from lattice reduction problems*, in Proceedings of CRYPTO 97, Lect. Notes in Comp. Sci. 1294, 1997, 112–131.
8. A.J. Menezes and P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1996.
9. T. Okamoto. *Provably secure and practical identification schemes and corresponding signature schemes*, Advances in Cryptology—Crypto '92, Lecture Notes in Computer Science 740 (E.F. Brickell, ed.) Springer-Verlag, 1993, 31–53.
10. Rosing, M.: *Implementing Elliptic Curve Cryptography*. Manning Publications Greenwich, CT (1999).
11. C.-P. Schnorr. *Efficient identification and signatures for smart cards*, Advances in Cryptology—Crypto '89, Lecture Notes in Computer Science 435 (G. Brassard, ed), Springer-Verlag, 1990, 239–251.
12. J.H. Silverman, *Lattices, Cryptography, and the NTRU Cryptosystem*, Proceedings of a DIMACS Conference, January 10–14, 2000, American Mathematical Society, to appear.
13. J. Stern. *A new identification scheme based on syndrome decoding*, Advances in Cryptology—Crypto '93, Lecture Notes in Computer Science 773 (D. Stinson, ed.), Springer-Verlag, 1994, 13–21.
14. J. Stern. *Designing identification schemes with keys of short size*, Advances in Cryptology—Crypto '94, Lecture Notes in Computer Science 839 (Y.G. Desmedt, ed), Springer-Verlag, 1994, 164–173.

12 Jeffrey Hoffstein, Joseph H. Silverman

15. J. Stern, *Lattices and Cryptography: An Overview* in Public Key Cryptography (PKC '98), Lecture Notes in Computer Science 1431, Springer-Verlag, 1998, 50–54.
16. D. Stinson, *Cryptography: Theory and Practice*. CRC Press, 1997.