SANDIA REPORT
SAND2000-2766
Unlimited Release
Printed November 2000

Low-Power Public Key Cryptography

Cheryl Beaver, Timothy Draelos, Victoria Hamilton, Richard Schroeppel,
Rita Gonzales, Russell Miller, and Edward Thomas

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND2000-2766
Unlimited Release
Printed November 2000

Low-Power Public Key Cryptography

Cheryl Beaver, Timothy Draelos, Victoria Hamilton, and Richard Schroeppel
Cryptography and Information Systems Surety Department

Rita Gonzales and Russell Miller
Digital Microelectronics Department

Edward Thomas
Statistics & Human Factors Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0449

Abstract

This report presents research on public key, digital signature algorithms for cryptographic authentication in low-powered, low-computation environments. We assessed algorithms for suitability based on their signature size, and computation and storage requirements. We evaluated a variety of general purpose and special purpose computing platforms to address issues such as memory, voltage requirements, and special functionality for low-powered applications. In addition, we examined custom design platforms. We found that a custom design offers the most flexibility and can be optimized for specific algorithms. Furthermore, the entire platform can exist on a single Application Specific Integrated Circuit (ASIC) or can be integrated with commercially available components to produce the desired computing platform.

Several digital signature algorithms are candidates for low-power usage, but some may have restrictions. We recommend an elliptic curve implementation of an El Gamal signature as a general solution. We used special elliptic curve and finite field operations to optimize the algorithm, and designed an implementation for an ASIC. The design is available to interested readers and is ready for easy integration into specific applications.

This page intentionally left blank.

Contents

Figures

Tables

This page intentionally left blank.

1. Introduction

Cryptography is used to provide privacy, integrity, authentication, and non-repudiation of data. There are two basic categories of cryptography today: conventional or symmetric key cryptography, and public or asymmetric key cryptography. Both conventional and public key cryptography provide privacy and integrity of data through encryption; however, only public key cryptography provides the functions of authentication and non-repudiation. This is done through the use of digital signatures.

Public key, digital signature algorithms are typically computationally intensive and generate large signatures. Our focus is on low-power environments where available computing resources and power are limited. Such environments may be too prohibitive to support digital signatures. Yet, such environments often must support devices transmitting critical data that must be authenticated. For example, multilateral treaties may require remote monitoring devices for treaty verification. The device may be located in hostile territory; it may use only a small processor; and it may be required to run off a single battery that is left unattended for years. The data reported by the device will have international ramifications and must be authenticated. Digital signatures would be ideal in such situations, yet the demands of a public key algorithm may not conform to the environment.

The purpose of this Laboratory Directed Research and Development Project (LDRD) was to find, modify, or invent a computationally inexpensive public key signature scheme for efficient use in such low-power environments. We researched and tested implementations of several different types of algorithms and also investigated commercial, special purpose, and custom hardware for efficiency.

As a general solution, we recommend an elliptic curve-based digital signature scheme optimized for low-power use on custom hardware. We developed and tested such a design targeted to a custom integrated circuit (ASIC, FPGA, PLA) for use in low-power public key cryptography applications. Even more efficient algorithms may be available for use in special situations, and we point those out as well.

In symmetric key cryptography, every user of the system shares a common key that is used both for encryption and decryption. The algorithms are fast and efficient, and use commands easy for computers to carry out, such as *shift* and *exclusive-or* operations. The encryption functions are effectively "one way" (i.e., there is no mathematical way to invert the operation); however, if one has access to the key, the encryption process can be reversed to decrypt.

Since the encryption/decryption key must be kept secret and shared by users in several locations, key management problems can arise. In public key cryptography, each participant holds a private-public key pair and only the user's private key must be kept secret. Although the public key is mathematically related to the private key, knowledge of the public key reveals nothing about the secret private key and so the public key can be openly available to all. The public key is used to encrypt data that only the holder of the private key can decrypt. Because of the asymmetry of the keys and the fact that a private key belongs to a unique user, the private key can also be used to generate a digital signature unique to the user and each message.

Although key asymmetry in public key cryptography enables cryptographic functionality not possible with symmetric cryptographic systems, these extra capabilities are not "free." In order to enable different keys for encryption and decryption, algorithms with special mathematical properties must be used. Exploitation of these mathematical properties leads to attacks on the system. To protect against such attacks, the keys that must be used are much larger than those used with symmetric key cryptosystems. The mathematical operations, together with the large keys, means the work involved in a public key system is usually many times greater than with conventional cryptography.

Our goal was to find, modify, or create a public key signature scheme that could be adapted to perform in a low computation, low-power environment. There were several issues to consider in each scheme, including signature size and computation and storage requirements.

- **Signature Size:** Many of the signature schemes require that large signatures be sent with the message, regardless of message size. Since bandwidth can consume the most power in the system, minimizing the size of the signature is important.

- **Computation**: Most public key systems require modular exponentiation, a very costly operation. One of the first steps was to look at systems in which modular exponentiation was minimized or could be avoided in the signing function of the scheme (i.e., systems for which pre-computation is possible).

- **Storage**: Although storage may be a lesser issue than power consumption, unlimited storage should not be assumed.

- **RAM/ROM**: Although RAM/ROM size may be a lesser issue than power consumption, unlimited RAM/ROM should not be assumed.

Each algorithm in a public key system has a "hard" mathematical problem as the basis of its security. Used in this sense, "hard" means the operation is computationally not feasible to invert without special knowledge (e.g., the private key). We looked at algorithms based on a variety of problems including number theoretic, probabilistic, coding theoretic, polynomial-based, and hash-based. While several algorithms have great potential, others perform well only in certain situations. The study of the algorithms, together with some optimizations and a comparison of their features, can be found in Section 2.

The secondary goal of this LDRD was to evaluate various computing platforms for suitability to low-power public key cryptography. While section 2 presents an analysis of candidate public key algorithms, sections 3 and 5 present information on various computing platforms on which these algorithms must run and section 4 discusses currently available memory options. We discuss basic power consumption issues and processing features that lend themselves to low-power public key cryptography applications as well as evaluate specific general-purpose commercially available processors as candidate computing platforms. Section 5 also offers information on special purpose (e.g., smart card) and custom (e.g., ASIC) computing platforms and their potential in low-power public key cryptography applications. For hardware applications, memory can be a vital issue. Indeed, we found that the performance of some algorithms improved dramatically if pre-computed stored values could be accessed during implementation.

Section 6 discusses low-power design techniques for a custom-computing platform. A custom-design platform offers the most flexibility in giving the customer exactly what is needed. Custom designs can address issues such as speed, power, security, and functionality. The entire computing platform can exist on a single Application Specific Integrated Circuit (ASIC), or can be integrated with commercially available components to produce the desired computing platform. Reconfigurable logic (Field Programmable Gate Arrays–FPGAs, Programmable Logic Arrays–PLAs) can be just as easily targeted as ASICs depending on the needs of the design.

Finally, having determined that an elliptic curve implementation of the El Gamal digital signature scheme has the most potential as a general-purpose solution, in Section 7, we present a design that further optimizes the code and takes advantage of the custom-computing platform. The design is targeted to a custom integrated circuit (ASIC, FPGA, PLA) for use in low-power public key cryptography applications.

2. Algorithms

Public key algorithms base their security on the improbability that an adversary will solve a particular problem. The problems may be difficult in the sense that they are thought to be NP-complete, or it may simply be the case that no efficient algorithms are known. Number theoretic problems provide the most common basis for public key schemes of today. Indeed, most cryptosystems base their security on the difficulty of solving the discrete logarithm problem, the integer factorization problem, or a related analog of these problems. These schemes involve expensive computations such as modular multiplications and can have long signatures. Hence, these algorithms may not be suitable for many low-power environments. Nevertheless, optimizations may be made to improve the performance of these algorithms.

In addition to studying common public key schemes, we broadened our search to include some lesser-studied and lesser-used algorithms in the hope they would be useful in a low-power environment. In particular, we studied schemes based on coding theoretic problems, probabilistic schemes, hash-based schemes and polynomial schemes. We found that coding theoretic schemes generally have very large keys, making them unsuitable for low-power environments. We present them here for informational purposes only.

Probabilistic schemes have been defined primarily as identification schemes. Most are based on NP-complete problems, and any public key identification scheme can be converted into a signature scheme. Although we had difficulty translating them into signature schemes efficient enough for the low-power world, those ideas are presented as a possible basis for future research.

Hash-based schemes are very efficient computationally but can be limiting in other ways. For example, many schemes require pre-determination of the number of signatures that will be signed. Others are based on chaining, and verification becomes impossible if messages are ever lost, leaving them open to denial of service attacks.

Finally, polynomial-based schemes hold promise for computing shorter signatures, but the public key sizes are very large and verification can be difficult.

In each category (number theoretic, coding theoretic, probabilistic, hash-based and polynomial) we examined a number of algorithms and analyzed their performance based on our criteria of signature size, storage and computation requirements. We present a comparative summary of the best candidate algorithms in a table near the end of this section. Some of the algorithms are good for signatures and some are most efficient at verification. Often, the very low-power environments are concerned only with signing; hence, the emphasis is on signing.

If two-way authentication is desired, different algorithms may be used for signing and verification. The best general solution we found for signatures was the elliptic curve-based version of the El Gamal scheme. In addition to being quite efficient as is, we found some very helpful optimizations in the elliptic curve operations as well as the underlying field operations. The implementation of the algorithm in custom hardware design is described in Section 7.

All of the problems presented here are "hard" in the sense that there are no known algorithms for solving them efficiently. The recommended key sizes are chosen to protect against attacks that the best computers with the most efficient algorithms could reasonably mount. As algorithms to solve these problems become more efficient, key sizes should be increased. The user is advised to consult the literature for current recommended key sizes.

2.1 Number Theoretic Schemes

There are two main problems that the security of number theoretic schemes is based upon: the discrete logarithm problem and the integer factorization problem. Given a group G and two group elements α, $\beta=\alpha^t$, the discrete logarithm problem is to find the integer t. The group G should be chosen so that this problem is difficult. The most common groups chosen for use include $(Z/pz)^* = \{1,2,..,p-1\}$, the non-zero group of integers modulo a large (non-smooth) prime, p, and the group of points on an elliptic curve modulo p. Given a large integer n, the integer factorization problem is to find the divisors of n. Typically, cryptographic systems based on this problem use a large integer n which is the product of two large primes p and q: $n = pq$. Most of the algorithms presented here are based on one of these problems or a variant of one of these problems.

The following algorithms require modular multiple-precision operations. Although modular arithmetic usually requires a fair amount of computing power, these algorithms minimize these requirements by avoiding exponentiation or by minimizing the cost of exponentiation, and by using pre-computation and added storage. Features associated with each algorithm make it a viable choice despite the requirement for modular multiple-precision arithmetic operations.

2.1.1 ESIGN

The ESIGN (Efficient digital SIGNature) Algorithm is described in [MvV97]. It is based on the Integer Factorization problem. This algorithm may be a candidate in a low-power environment where two-way authentication is required.

To generate a key pair:

1. Select random primes p and q such that $p \geq q$ and p, q are roughly the same bit-length
2. Compute $n = p^2 q$
3. Select a positive integer $k \geq 4$

An entity's public key is (n, k). The corresponding private key is (p, q).

To generate a signature:

1. Compute $v = H(m), H : \{0,1\}^* \rightarrow Z/nZ$
2. Select a secret random integer $x, 0 \leq x < p$
3. Compute $w = \left| ((v - x^k) \bmod n)/(pq) \right|$ and $y = w \cdot (kx^{k-1})^{-1} \bmod p$
4. Compute the signature $s = (x + ypq) \bmod n$

To verify a signature:

1. Compute $u = s^k \bmod n$ and $z = H(m)$
2. Accept the signature *iff* $z \leq u \leq z + 2^{\left\lceil \frac{2}{3} \lg n \right\rceil}$

The recommended modulus length, n, is 768 bits. Therefore the signature length is also 768 bits or 96 bytes. Since k can be chosen to be small, the exponentiation required to generate w can be optimized for a small known exponent and a known modulus. The same is true of the exponentiation required for verification, making this algorithm a possibility in a low-power environment where two-way authenticated communication is required.

2.1.1.1 Evaluation of ESIGN

Table 2.1 Parameter Sizes for ESIGN

Parameter	Description	Size (in bits)
P	Secret prime	≈256
Q	Secret prime	≈ 256
N	Composite modulus of two primes p and q, $n = p^2 q$	≥ 768
K	Public integer ≥ 4	NA
X	Secret random integer, $0 \leq x \leq p-1$	≈256
V	Hash digest of the message	160
W	$\left\lvert ((v - x^k) \bmod n)/(pq) \right\rvert$	≈512

Parameter	Description	Size (in bits)
Y	$w \cdot (kx^{k-1})^{-1} \bmod p$	≈ 256
S	Signature, $(x + ypq) \bmod n$	≥ 768

Signature operations required: Random number generation mod p, a hash, a small exponentiation mod n, a small exponentiation mod p, a division by pq, and an inverse mod p

Verification operations required: A small exponentiation mod n, a hash, and a comparison

Storage required to sign (in bits): ≈ 256 bits for p, ≈ 512 bits for pq, and 768 bits for n

Amount of data transmitted: 96 bytes

2.1.2 Feige-Fiat-Shamir

The Feige-Fiat-Shamir Algorithm is described in [MvV97]. It is based on the intractability of computing square roots modulo n. Computing square root modulo n is equivalent to the integer factorization problem: if you know the factorization of n, then taking the square root of a number modulo n is easy; conversely if you know the square root of a numbers modulo n, you can find the factors of n easily. One of the advantages of the Feige-Fiat-Shamir scheme is that, unlike the RSA scheme, all entities in the system can share n. This may be convenient in a situation where there are many users. Of course, in this case, n must be generated by a TTP (Trusted Third Party) since p and q (the factors of n) must not be revealed to any entity.

To generate a key pair:

1. Generate n (which can be shared), a composite of two large secret primes p and q
2. Generate k distinct random integers $s_1, s_2, \ldots s_k \in (Z/nZ)^*$
3. For each $s_j, 1 \leq j \leq k$, compute $v_j = s_j^{-2} \bmod n$

An entity's public key is $(v_1, v_2, \ldots, v_k)$ and n. The entity's private key is $(s_1, s_2, \ldots s_k)$.

To generate a signature:

1. Generate a random integer $r \in (Z/nZ)^*$
2. Compute $u = r^2 \bmod n$
3. Compute a k-bit hash, $e = H(m \| u)$
4. Compute $s = r \cdot \prod_{j=1}^{k} s_j^{e_j} \bmod n$

5. The signature is (e,s)

To verify a signature:

1. Compute $w = s^2 \cdot \prod_{j=1}^{k} v_j^{e_j} \bmod n$
2. Compute $e' = H(m \| w)$
3. Accept the signature *iff* $e = e'$

The length of the modulus should be at least 768 bits, and the recommended value for k is 160. The length of e should be 160 bits. Therefore, signature generation requires, on average, 160/2 or 80 modular multiple-precision multiplications. The signature length under these recommendations is 160 + 768 bits or 116 bytes. The signature length is probably too long for this application.

2.1.2.1 Evaluation of Feige-Fiat-Shamir

Table 2.2 Parameter Sizes for Feige-Fiat-Shamir

Parameter	Description	Size (in bits)
N	Composite modulus of two primes p and q, $n=pq$	≥ 768
K	Cardinality of sets S and V	≥ 160
S	Set of private keys $s_i, 1 \le i \le k, 0 < s_i < n$	≥ 768
V	Set of public keys $v_i, 1 \le i \le k, 0 < v_i < n, v_i = s_i^{-2} \bmod n$	≥ 768
R	Random number $0 \le r \le n$	≥ 768
U	$r^2 \bmod n$	≥ 768
E	$H(m \| u)$	160
e_I	i^{th} bit of e, $0 \le i \le k-1$	NA
S	$r \cdot \prod_{j=1}^{k} s_j^{e_j} \bmod n$	≥ 768

Signature operations required: Random number generation, modular squaring, a hash, maximum of 160 modular multiplications (on average 80 modular multiplications)

Verification operations required: One modular squaring, a hash, maximum of 160 modular multiplications (on average 80 modular multiplications)

Storage required to sign (in bits): 768 bits for n and 768 bits for each of the 160 s_i

Amount of data transmitted: 116 bytes

2.1.3 Optimal El Gamal Scheme

The Optimal El Gamal-type Algorithm is described in [HX94] and [NR94]. It is based on the intractability of the discrete log problem. The prime p is chosen to be about 768 bits and q is defined to be a prime dividing $(p-1)$ of about 160 bits. The number g is an element of $(Z/pZ)^*$ of order q. The following equations specify the digital signature r and s:

$$r = (g^k \bmod p) \bmod q$$
$$s = (rxH(m) - k) \bmod q$$

Here, x constitutes the private key and k is a per message secret. $H(m)$ is the message digest. The k, r, and rx values can be pre-computed and used for each signature generation. Only a single modular multiple-precision subtract and multiply are required to compute the second part of the signature, s. The signature length is 40 bytes. Each of the pre-computed values is 20 bytes in length. The implementation would have to store three 20-byte pre-computed values for each message it is to sign. When those values have been used, the application could either no longer sign messages or could use some other mechanism to sign messages.

Example: (3*20 bytes) * (1 message/day) * (365 days/year) * (5 years) = 109,500 bytes of pre-computed data

Another disadvantage is that the pre-computed values must remain secret since they are all comprised of x (the private key) and/or k (the per message random value). Since the low-power device will not be computing these values, they must be protected wherever they are generated and during transmission to the low-power device. In addition, signature verification is not optimized:

$$y^{rH(m)} = r' g^s \bmod p$$
$$r' = ((y^{rH(m)} g^{-s}) \bmod p) \bmod q$$
$$r' = ((y^{rH(m)} g^{q-s}) \bmod p) \bmod q$$

Does $r = r'$? If *yes*, accept; if *no*, reject

Therefore, two-way authenticated communications using this method alone are not suitable for the low-power environment. However, it may be possible to authenticate using one mechanism in one direction and verify using another mechanism in the opposite direction.

This pre-computational technique can be used for any of the El Gamal signature schemes including DSA. In addition, one of the values required for the modular multiplication can be stored in Montgomery form to allow concurrent multiplication and modular reduction rather than

a multiplication followed by reduction. The Montgomery form of a value requires no additional memory for storage.

2.1.3.1 Evaluation of Optimal El Gamal Scheme

Table 2.3 Parameter Sizes for Optimal El Gamal

Parameter	Description	Size (in bits)
P	Prime modulus	≥ 768
Q	Prime divisor of $p-1$	≥ 160
G	Any value such that the order of g is q	≥ 768
X	Secret key	160
Y	$g^x \bmod p$	≥ 768
H(m)	Hash digest of the message	160
K	Random per message secret	160
R	$(g^k \bmod p) \bmod q$	160
S	$(rxH(m)-k) \bmod q$	160

Signature operations required: Hash of message, 160-bit modular multiplication and subtraction

Verification operations required: Hash of message, 160-bit modular multiplication, two modular exponentiations in the size of p

Storage required to sign (in bits): 768 ≤ p, g, 160 bits for q, and (3*160) bits for each message to be signed

Amount of data transmitted: 40 bytes

The Balanced El Gamal Scheme, a variation of the Optimal El Gamal scheme, requires less storage and only two more modular multiplications to sign. See Appendix A, Section 9.1.1 for more details.

Note: Some of the algorithms described above take advantage of pre-processing to improve efficiency, at the cost of some storage. Care must be taken that the pre-processing does not affect the security of the system. The Schnorr scheme, described in [S89], is a DSA variant that takes advantage of a pre-processing mechanism. However, the pre-processing mechanism presented by Schnorr has been successfully attacked, as described in [dR93]. The algorithm is described in Appendix A, Section 9.1.2.

2.1.4 Elliptic Curve El Gamal

Elliptic curves consist of pairs (x,y) (points), which are solutions to a certain cubic equation together with a distinguished point, O, called the origin (point at infinity). For use in

cryptography, we consider only those points whose coordinates lie in some finite Galois field. Typically, the field is either $GF(p)$ for some prime p, or $GF(2^n)$. For a general introduction to elliptic curves, see [Sil86]. For our purposes, we consider the case where the field of definition for the points on the curve is $GF(2^n)$. The arithmetic is faster and, in that case, the curve E can be given by an equation of the following form: $E: y^2 + xy = x^3 + ax^2 + b$, with $a,b \in GF(2^n)$.

There is an addition law that can be defined on the points of the curve. Suppose $M_1 = (x_1, y_1)$ and $M_2 = (x_2, y_2)$ are two points on the curve E. Then, the addition of the points, $M_1 + M_2 = M_3$ is defined geometrically (see Fig. 2.1): Draw a line through the points M_1 and M_2. Since it is a cubic, the line will intersect the elliptic curve at exactly one other point, P. The point $M_3 = M_1 + M_2$ is the point on the curve defined by the reflection of P about the x-axis. For an algebraic description of the addition law, see [P1363]. The set of points on the curve defined over the finite field $GF(2^n)$ form a group with identity element O. Multiplication is defined on E as repeated addition: $nP = P + P + ...P$ (n times). The discrete logarithm problem is defined on the elliptic curve group as follows: Given an elliptic curve E and points $G, V = nG \in E$, find n. Note that multiplication in the elliptic curve group is analogous to exponentiation in a finite field.

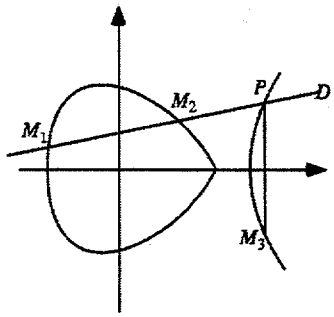

Figure 2.1 Addition on an Elliptic Curve

Any discrete logarithm-based cryptosystem can be carried out using the group of points on an elliptic curve instead of the usual finite field group. However, traditional attacks on discrete log-based systems over finite fields do not carry over when the base group is an elliptic curve. For this reason, smaller key sizes can be used and the algorithms are often more efficient. We describe the El Gamal Signature scheme using elliptic curves.

To generate a key pair and system parameters for a curve over the field $GF(2^m)$ when m is around 160 bits:

1. Choose an elliptic curve E: $y^2 + xy = x^3 + ax^2 + b$ with coefficients $a,b \in GF(2^m)$. We note that a can always be determined by a single bit. Furthermore, the coefficient b (implied by the coordinates of the points) is never used in any algorithms, so the storage required for the coefficients of the elliptic curve is really just one bit

2. Choose r, a prime divisor of the order N of the elliptic curve, and let k be such that $rk = N$.

3. Choose $G = (g_1, g_2)$, a point on the elliptic curve of order r

4. Choose the long-term private key s

5. Compute the public key W = sG

To generate a signature on a message M:

1. Generate a key pair (u, $V = uG$), where u is a random integer mod r. Let $V = (x_V, y_V)$ ($V \neq$ O because V is a public key)

2. Convert x_V into an integer i

3. Compute an integer $c = i \bmod r$. If $c = 0$, then go to Step 1

4. Let f = Hash(M). Compute an integer $d = (cfs + u) \bmod r$. If $d = 0$, then go to Step 1

5. Output the pair (c, d) as the signature

To verify a signature:

1. If c is not in $[1, r-1]$ or d is not in $[1, r-1]$, output "invalid" and stop

2. Compute integer $h = cf \bmod r$

3. Compute an elliptic curve point $P = hW - dG$. If P = O, output "invalid" and stop. Otherwise, $P = (x_P, y_P)$

4. Convert the field element x_P to an integer i

5. Compute an integer $c' = i \bmod r$

6. If $c' = c$, then accept

2.1.4.1 Evaluation of Elliptic Curve El Gamal over *GF*(2^m)

Table 2.4 Parameter Sizes for Elliptic Curve El Gamal over *GF*(2^m)

Parameter	Description	Size (in bits)
M	Size of binary finite field	≈160
A,b	Coefficients for the elliptic curve $y^2 + xy = x^3 + ax^2 + b$ (storage for value of a only)	≈1

Parameter	Description	Size (in bits)
R	Prime divisor of the order of the elliptic curve	≈160
G	Point of the elliptic curve of order q	≈320
S	Secret random integer, private key	≈160
F = H (m)	Hash digest of the message	160
W	Public point $W = sG$ public key	≈320
U	Random value mod r	≈160
C	x-coordinate of $V = uG$ mod r	≈160
D	$d = (cfs+u) \mod r$	≈160

Signature operations required: Hash, an elliptic curve scalar multiply, two modular multiple-precision multiplies and an addition modulo r

Verification operations required: Hash, one multiplication mod r, an inverse, two elliptic curve scalar multiplications, and one point addition

Storage required to sign (in bits): 320 bits for r and s, 160 bits for u, $d = (cfs+u) \mod r$, 1 bit for a, and 320 bits for G

Amount of data transmitted: 40 bytes

2.1.5 Knapsack Schemes

Knapsack schemes are based on the subset sum problem. The subset sum problem is defined in [MvV97]:

Given: a set $\{a_1, a_2, ..., a_n\}$ of positive integers called a knapsack set, and a positive integer s

Determine: whether or not there is a subset of the a_j that sum to s (i.e., determine whether there exists $x_i \in \{0,1\}, 1 \le n \le n$, such that $\sum_{i=1}^{n} a_i x_i = s$).

This problem is known to be NP-complete and the computational version is known to be NP-hard.

To use the subset sum problem as the basis of a public key scheme, an instance of the subset sum problem which is easy to solve is selected and transformed into an instance of the subset sum problem which is difficult to solve. The first instance can serve as the private key and the second instance can serve as the public key.

The Chor-Rivest scheme is the only known knapsack public key scheme that has not been broken. Unfortunately, using recommended parameter sizes, the public key is roughly 40,000 bits in length, making this algorithm not feasible for the low-power environment. We give a description in Appendix A, Section 9.1.3 for informational purposes.

2.2 Incremental Schemes

Incremental cryptography is described in [BGG94]. The idea behind incremental cryptography is that once having signed a particular message *m*, the work necessary to sign a message *m'*, which is a modification to *m*, should be proportional to the difference between the two messages. Thus, a small change should require far less effort than signing a completely different message. [BGG94] use the standard mechanism of hashing and then transforming to compute a digital signature. They propose that what is needed is an incremental collision-free hash algorithm that would be applied only to the blocks of the message which had changed. The transformation function would, in fact, remain the same as with standard digital signature mechanisms.

There are two problems associated with this proposed mechanism. The most serious problem is a result of the fact that the only known incremental collision-free hash algorithm is based on *n* exponentiations modulo a *k*-bit prime where *k* is the size of each message block. Therefore, at least one modular exponentiation must be performed in addition to any exponentiation(s) already performed as part of the transformation. Except for large messages, standard hash algorithms are generally much more computationally efficient than even a single modular exponentiation. In addition, incremental schemes still require exponentiation(s) for transformation of the hash result. The scheme described by [BGG94] is more computationally intensive than standard public key signature schemes.

2.3 Coding Theoretic Schemes

There are several public key schemes that are based on coding theory. In general, the keys for these schemes can be quite large, which makes them not feasible for the low-resource computing environment.

The McEliece Public Key Encryption Algorithm is described in [MvV97]. It is based on the difficulty of decoding an arbitrary linear code which is known to be NP-hard. It has received little practical attention due to the size requirements of public keys. Since the size of the private key is 264 kilobytes, we do not consider it a candidate for low-power computing, but we do present a description in Appendix A, Section 9.1.4 for informational purposes.

2.4 Probabilistic Identification Schemes

Probabilistic schemes have been defined primarily as identification schemes. Identification schemes give interactive zero-knowledge proofs of identity via a challenge-response protocol. Any public key identification scheme can be converted to a public key signature scheme via the use of a cryptographically secure hash function to simulate the random challenges. There are several problems (typically NP-hard) that can provide a basis for identification schemes. Some examples are the permuted kernel problem, syndrome decoding for error-correcting codes, the constrained linear equation problem and the perceptrons problem.

When the algorithm is translated from an identification scheme to a signature scheme, the challenges that must be simulated cause a large communication overhead. If these are to be used in a low-power environment, modifications must be made to reduce the overhead. We attempted

such modifications but ran into some stumbling blocks. We describe below some of the identification schemes and then give our insights on how they may be efficiently translated into signature schemes based on our attempts.

2.4.1 Permuted Kernel Problem Scheme

This public key identification scheme is defined in the abstract [S89a], with further analysis done in [BCCG92] and [PC93]. It is based on an NP-complete algebraic problem known as the permuted kernel problem. The problem is defined as [MvV97]:

Given: an $m \times n$ matrix A over Z/pZ, p prime and relatively small (e.g., 251), and an n-vector V

Find: a permutation π on $\{1, ..., n\}$ such that $V_\pi \in Ker(A)$

Where: $Ker(A)$ is defined as the kernel of A consisting of all n-vectors W such that $AW = [0 \ldots 0] \bmod p$

Public Information: A, p, V

Private Identification Key: π

Use in a Three-Pass Zero Knowledge Identification Scheme (Shamir):

1. The prover (A) chooses a random n-vector R and a random permutation σ, and sends the cryptographically hashed values of the pairs (σ, AR) and $(\pi\sigma, R_\sigma)$ to the verifier (B)

2. B chooses a random value $0 \le c < p$, and asks that A send $W = R_\sigma + c(V_\pi)_\sigma$

3. After receiving W, B asks A to reveal either σ or $\pi\sigma$. In the first case, B checks that $(\sigma, A_\sigma W)$ hashes to the first given value, and, in the second case, B checks that $(\pi\sigma, W - c(V_\pi)_\sigma)$ hashes to the second given value

Note:

$$A_\sigma W = A_\sigma (R_\sigma + c(V_\pi)_\sigma) = A(R + cV_\pi) = AR$$
$$W - c(V_\pi)_\sigma = R_\sigma$$

The probability that a cheater can evade detection is 1/2, so the protocol is repeated k times to reduce the probability of successful cheating to some acceptable limit, i.e., $1/2^k$. If a prover can pass the test, his identity is accepted. We attempted to modify this identification scheme into a signature scheme suitable to the low-power environment. A log of our progress can be found in Appendix B, Section 10.

2.4.2 Syndrome Decoding Based Identification Scheme

This public key identification scheme is defined in [S93]. This scheme is based on the syndrome decoding problem for error-correcting codes. The security of this scheme is based on the hardness of decoding a word of given syndrome with respect to some binary linear error-correcting code.

2.4.3 Constrained Linear Equation Scheme

This public key identification scheme is described in [S94]. This scheme is based on a combinatorial problem known as the Constrained Linear Equation (CLE) problem. It consists of solving a set of linear equations modulo some small prime q, where the unknowns belong to a specific subset of the integers modulo q. A CLE problem can be defined as

- Given: A small prime number q, a system S of r homogeneous linear equations with k unknowns whose coefficients are integers mod q, and a subset k of the integers mod q
- Find: A solution S consisting of k elements of the given set X

It is easily seen that the problem is NP-complete. It is further assumed that the CLE is intractable in the sense that no probabilistic polynomial time algorithm can take as its input the values q, S, X, and output, with non-negligible probability, a solution of S consisting of k elements of the given set X.

2.4.4 Perceptron Scheme

Several difficult, indeed, NP-complete, problems exist in the field of machine learning. An identification scheme based on an NP-complete problem faced in machine learning, called the perceptrons problem, is presented in [P95]. The perceptron problem can be defined as:

- An ε-vector or matrix is a vector or matrix whose components are either -1 or $+1$
- Given: an ε-matrix A of size (m x n)
- Find: an ε-vector Y of size n such that $AY \geq 0$

The permuted perceptron problem can be defined as:

- Given: an ε-matrix A of size (m x n) and a multiset (a set where repeated elements are allowed) S of non-negative integers of size m
- Find: an ε-vector Y of size n such that $\{\{(AY)_j | j = \{1,\ldots,m\}\}\} = S$

All of these problems give rise to zero-knowledge identification schemes. They can also be translated into signature schemes, but the overhead involved is huge. In the following subsection,

we give suggestions on how to improve efficiency when converting from a zero-knowledge to signature scheme.

2.4.5 General Strategies For Efficiently Implementing Probabilistic Schemes

The following strategies may be useful in converting zero-knowledge schemes to signature schemes useful for the low-powered environment.

- The verifier pre-computes entities that otherwise would need to be transmitted, so that the signature length will be reduced at the expense of increased verification operations. For example, the verifier computes $(c_i, \tau_i, b_i)\ \forall_i$ in permuted kernel identification scheme (see Appendix B, Section 10.1).

- Reduce signature length by transmitting a permutation seed rather than the complete permutation. The seed is input to an algorithm that produces a permutation. Transmission requirements are reduced at the expense of increased verification operations. Permutations are used in the PKP and Syndrome Decoding algorithms.

- Reduce signature length by using only a subset of "commitment bits." For example, consider the Syndrome Decoding case (see Appendix B, Section 10.2). Let

  $$C_j = \{c_1^j, c_2^j, c_3^j\}, j = 1,2,\ldots,k$$

  represent the k sets of commitments that are required. Each commitment (c_i^j) assumed to be the output of a hash function (typically 64 to 128 bits). Suppose that instead of making a full commitment, the prover commits to only the first q bits, thus reducing the signature size at the expense of reduced security. In the case of Syndrome Decoding, the security is reduced from about

  $$\left(\frac{2}{3}\right)^k \text{ to } \left(\frac{2}{3}+\frac{1}{3}\cdot\left(\frac{1}{2}\right)^q\right)^k.$$

  The following table illustrates how the security varies with q and k.

Table 2.5 Security versus q and k for Syndrome Decoding

	$k = 20$	$k = 30$	$k = 40$	$k = 60$
$q = 1$	.026	.0042	.00068	.000018
$q = 2$	.0032	.00018	.00001	$3\text{x}10^{-8}$
$q = 3$	.001	$3.2\text{x}10^{-5}$	$1\text{x}10^{-6}$	$1\text{x}10^{-9}$
$q = 4$	.0056	$1.3\text{x}10^{-5}$	$3\text{x}10^{-7}$	$2\text{x}10^{-10}$
Large q	.0003	$5.2\text{x}10^{-6}$	$9\text{x}10^{-8}$	$3\text{x}10^{-11}$

In this case, the total number of commitment bits transmitted is $3 \cdot q \cdot k$. (Note that the commitment bits form only part of the total signature. For example, if our desired security was 1 x 10^{-6}, we could obtain this in various ways. One way would be to select $q = 3$ and $k = 40$ for a total of 360 "commitment bits." One can also use this table to establish the trade-off between q and security for a particular value for k.

In the case of the PKP Algorithm, $C_j = \{c_1^j, c_2^j\}, j = 1,2,\ldots,k$. Using an analogous approach, the security of this scheme is reduced from about

$$\left(\frac{1}{2}\right)^k \text{ to } \left(\frac{1}{2}+\left(\frac{1}{2}\right)^{q+1}\right)^k.$$

2.5 Hash-Based Signature Schemes

Signatures based on hash functions can be less computationally intensive than their mathematical counterparts, and so are worth looking at for a low-power environment. We start our investigation of hash-based signature schemes by looking at some one-time signature schemes. Any one-time scheme can be extended to an N-time scheme, but this is usually done naively and the corresponding signatures can become unreasonably long. Furthermore, in an N-time scheme, only N messages can be signed. The larger N is, the larger the signature and the memory required to construct a signature. If there are a very large or unknown number of messages to sign, it may be impractical to pre-determine a number (N) of messages to sign. We will investigate an alternative to the N-time schemes whereby the signature for a message is based upon corroborating information in previous messages. These schemes can be computationally efficient and are worth consideration if only a finite (pre-determined) number of messages must be signed or if there is little possibility that messages may be lost.

2.5.1 One-Time Signatures

One-time digital signature schemes can be used to sign only one message. Otherwise, forgery is possible. A new key pair is required for each message. However, one-time signature schemes can be very efficient and, given that there is a mechanism for authenticating the necessary public-information, can be converted to generalized signature schemes.

2.5.1.1 Lamport's Scheme

The Lamport one-time signature scheme essentially signs one bit of the message at a time. Let m be the length of the message to sign. Let h be a hash function which outputs k bits.

To generate a key pair:

1. Generate $2m$ random strings $A_1,\ldots,A_m$, $B_1,\ldots, B_m$ each of length k bits
2. The public key is $(X_1,\ldots,X_m,Y_1,\ldots,Y_m)$ where $X_j = h(A_j)$ *and* $Y_j = h(B_j)$

3. The private key is $(A_1,...,A_m, B_1,...,B_m)$

To generate a signature:

> The signature of an m-bit message $M = b_1...b_m$ is $(S_1,...,S_m)$ where $S_j = A_j$ if $b_j = 0$ and $S_j = B_j$ if $b_j = 1$

To verify a signature:

> Check that $h(S_j) = X_j$ if $b_j = 0$ and $h(S_j) = Y_j$ if $b_j = 1$

The Signature and Verification algorithms are quite easy; however, the length of the signature is $k*m$ bits. This can be quite large if the message is long. There are two very similar variations of this scheme, both of which eliminate the need to 'sign' the zero bits. The first is the Lamport/2 scheme, which cuts the length of the signature essentially in half. The second is the Merkle scheme. There are further variations of each of these schemes where more than one bit is signed at a time.

2.5.1.1.1 The Lamport/2 Scheme

The Lamport/2 Scheme cuts the number bits in the signature approximately in half. Denote by m the length of the message to be signed and let $n = log\,[m/2] + 1$.

Key generation:

1. Choose $m + n + 2$ random k-bit strings: $A_1,...,A_m, B_1,...,B_n, C_0, C_1$
2. The public key is $(X_1,...,X_m, Y_1,...,Y_n, Z_0, Z_1)$ where $X_j = h(A_j)$ and $Y_j = h(B_j)$, and $Z_j = h(C_j)$
3. The private key is $(A_1,...,A_m, B_1,...,B_n, C_0, C_1)$

To generate a signature for $M = b_1...b_m$:

1. If more than $m/2$ bits of M are 0, then complement each bit of M and set $d = 1$ (we will denote this complemented version of M by M also); else leave M as it is and set $d = 0$
2. Denote the number of zeroes in M by $e = e_1...e_n$. Note that M has, at most, $m/2$ zeroes, and so the binary representation of e has at most n bits
3. Finally, denote the empty string by ε. The signature for $M = b_1...b_m$ *is* $(S_1,...,S_m, T_1, ..., T_n, U)$ where $S_j = A_j$ if $b_j = 0$ and $S_j = \varepsilon$ if $b_j = 1$; $T_j = B_j$ if $e_j = 0$ and $T_j = \varepsilon$ if $e_j = 1$; and $U = C_d$

To verify the signature:

1. Compare $h(U)$ to Z_0 and Z_1 to determine if $d = 0$ or $d = 1$. If $d = 1$, then complement M

2. Now, as in the general Lamport scheme, check to see if the appropriate pre-images (those of the zeroes) have been correctly supplied in the signature as well as those for the binary representation of e

3. If so, the signature verifies

2.5.1.1.2 The Partitioned Lamport/2 Scheme

In the Partitioned Lamport/2 scheme, the bits of the message are partitioned and so we effectively sign multiple bits at a time. As usual, denote by m the number of bits in the message to be signed. Let q be a small integer (say $q = 4,5,$ or 6) (Note that if $q = 1$, this is the same as the Lamport/2 scheme). Define also the following quantities:

- $r = [m/q]$
- $s = log([r(2^q-1)/2]+1)$
- $t = [s/q]$

Key generation:

1. Select $r + t + 2$ random k-bit integers: $A_1,...,A_r, B_1,...,B_t, C_0, C_1$
2. The private key is $(A_1,...,A_r, B_1,...,B_t, C_0, C_1)$
3. The corresponding public key is $X_j = h^{2q-1}(A_j)$ and $Y_j = h^{2q-1}(B_j)$, and $Z_j = h(C_j)$ where h^x denotes repeating the hash function x times

To generate a signature for $M = b_1...b_m$:

1. Partition $M = b_1...b_m$ into r groups of q bits each
2. Let n_j denote the integer value of the j^{th} group of bits (e.g., if the j^{th} group of bits is 1010, then $n_j = 10$)
3. Define $n = \sum n_j$. If $n < r(2^q - 1)/2$
4. Replace each n_j with $2^q - 1 - n_j$ and set $d = 1$; else set $d = 0$. Recompute n (if necessary)

   Note: This is done to force $n \geq r(2^q-1)/2$. Define $e = r(2^q - 1) - n$. Our construction of n forces e to have a binary representation $e = e_1...e_s$ of fewer than s bits. Finally, partition e into t groups of q bits each and denote by m_j the integer value of the j^{th} group of bits.

5. The signature of M is $(S_1, ..., S_r, T_1, ..., T_t, U)$ where $S_j = h^{n_j}(A_j), T_j = h^{m_j}(B_j), U = C_d$

To verify the signature:

1. Compute d by comparing U with $h(C_0)$ and $h(C_1)$

2. Check that the correct pre-images were supplied by hashing the values an appropriate number of times

Note on block size: It seems that the block size q into which the message is partitioned can be larger without effecting the security of the system. If q is larger, then the signature is smaller, however the number of hashes required to generate the public key and the signature grows exponentially since number of times the hash function needs to be performed is approximately the size of q. Furthermore, the number of hashes required in the verification process grows exponentially with q as well.

2.5.1.2 Merkle's Scheme

The Merkle one-time signature scheme can be converted to a generalized signature scheme using authentication trees, which are not described here. In addition, there are methods for reducing the size of the private key and improving the general efficiency.

2.5.1.2.1 Merkle's Single Bit Scheme

In general, Merkle's scheme was designed to sign a single bit of a message at a time. A basic description follows.

To generate a key pair:

1. Generate $t = n + \lfloor \lg n \rfloor + 1$ where n is the number bits in the message to be signed
2. Select random secret strings $k_1, k_2, \ldots k_t$ each of bitlength l
3. Compute $v_i = H(k_i), 1 \le i \le t$, where H is a pre-image-resistant hash function $H: \{0,1\}^* \rightarrow \{0,1\}^l$
4. The public key is $(v_1, v_2, \ldots, v_t)$. The corresponding private key is $(k_1, k_2, \ldots, k_t)$

To generate a signature:

1. Compute c, the binary representation for the number of zeroes in m
2. Form $w = m \| c = (a_1, a_2, \ldots, a_t)$
3. Determine the coordinate positions $i_1 < i_2 < \ldots < i_u$ in w such that $a_{i_j} = 1, 1 \le j \le u$
4. Let $s_j = k_{i_j}, 1 \le j \le u$
5. The signature is $(s_1, s_2, \ldots, s_u)$

To verify a signature:

1. Compute c, the binary representation for the number of zeroes in m
2. Form $w = m \| c = (a_1, a_2, \ldots, a_t)$
3. Determine the coordinate positions $i_1 < i_2 < \ldots < i_u$ in w such that $a_{i_j} = 1, 1 \le j \le u$
4. Accept the signature *iff* $v_{i_j} = H(s_j) \forall 1 \le j \le u$

Signature generation requires almost no computation. Signature verification is also quite efficient requiring fewer than $n + [\lg n] + 1$ hash operations, which makes this scheme attractive when two-way authentication is required. If $n = 128$ and $l = 64$, then the public and private keys each require 1088 bytes. The signature requires 600 bytes, which is quite long. In addition, to use Merkle's scheme as a generalized authentication scheme, key pairs must be pre-computed and stored. When these key pairs are exhausted, another signature mechanism must be used or the keys must be replenished.

Example: 1088 bytes/key * (1 message/day) * (365 days/year) * (5 years) = 1985600 bytes of pre-computed data.

2.5.1.2.1.1 Evaluation of Merkle's Single Bit Scheme

Table 2.6 Parameter Sizes for Merkle's Single Bit Scheme

Parameter	Description	Size (in bits)
N	Number of bits in the message	128, but may vary
L	Bit length of each k_i and v_i	64
T	$n + \lfloor \lg n \rfloor + 1$	136, but may vary
k_I	Random secret strings, $1 \le i \le t$	64
v_I	Hash of the secret strings, $1 \le i \le t$	64, but may vary depending on hash function used
C	Number of zeroes in the binary representation of the message, $0 \le c \le 128$	8
W	Message concatenated with c	136

Signature operations required: Compute the number of zeroes in the message (c), concatenate the message and this value (w). For each bit of w which is one; send the equivalent k_i

Verification operations required:	Compute the number of zeroes in the message (c), concatenate the message and this value (w). For each bit of w which is one, verify the equivalent public key v_i
Storage required to sign (in bits):	1088 bytes for all the k_i
Amount of data transmitted:	0–1088 bytes, depending on the number of ones in w

2.5.1.2.2 Merkle's Multiple Bit Scheme

Merkle's single bit scheme can be modified to sign multiple bits of the message at once. These modifications decrease the size of the keys and the signature.

Let

k = word size of the machine

$|m|$ = length of message to be signed

$t = |m|/k$

$r = \lceil (\lfloor \log t \rfloor + 1 + k)/k \rceil$

To generate a key pair:

1. Generate $t+r$ random bit strings $k_1, k_2, \ldots, k_{t+r}, |k_i| = l, 1 \le i \le t+r$
2. Compute $v_i = H(k_i), 1 \le i \le t+r$, where H is a pre-image-resistant hash function $H: \{0,1\}^* \rightarrow \{0,1\}^l$
3. The public key is $(v_1, v_2, \ldots, v_{t+r})$. The corresponding private key is $(k_1, k_2, \ldots, k_{t+r})$

To generate a signature:

1. Let the length of the m be kt bits
2. Write $m = m_1 \| m_2 \| \ldots \| m_t$, where each m_i is k bits long and represents a number between zero and $2^k - 1$ (i.e, unsigned).

3. Define

$$U = \sum_{i=1}^{t} (2^k - m_i) \leq t2^k$$

,

U can be represented in $\lg U \leq \lfloor \lg t \rfloor + 1 + k$ bits and can be written as $u_1 \| u_2 \| \ldots \| u_r$

4. The signature for m is

$$(s_1, s_2, \ldots, s_{t+r})$$

where

$$s_i = h^{m_i}(k_i), 1 \leq i \leq t$$
$$s_{t+i} = h^{u_i}(k_{t+i}), 1 \leq i \leq r$$

and h^c denotes a c-fold composition of h with itself

The signature is verified in a similar manner as in section 2.4.1.1, with the appropriate modifications. When these key pairs are exhausted, another signature mechanism must be used or the keys must be replenished.

2.5.1.2.2.1 Evaluation of Merkle's Multiple Bit Scheme

If the length of m is 128 bits and each k_i is 16 bits in length, then

- $t = 8$
- $r = \lceil (3 + 1 + 16)/16 \rceil = 2$
- $t + r = 10$
- Let each s_i be 128 bits long

The total signature length is 1280 bits, or 160 bytes, or four times as long as a DSA signature. The amount of storage required per message for all $k_i, 1 \leq i \leq t + r$ is (2 x (2 + 8)) or 20 bytes.

Example: (20 bytes/key) * (1 message/day) * (365 days/year) * (5 years) = 36500 bytes of pre-computed data.

Note on block size: The remark on block size after the Partitioned Lamport/2 scheme (Section 2.5.1.1.2) applies here as well.

2.5.2 Converting One-time Schemes to N-time Schemes

One simple way to convert a one-time scheme to an N-time scheme is to form a set of N private one-time keys and the corresponding public keys with the idea of using each of them once. The public key is just the concatenation of all the one-time public keys and likewise for the private key. A major drawback to this is that the size of the keys is very big.

An alternative to the above approach is to construct a hash tree. In this scheme, the signer sets up the keys for N one-time schemes using a common hash function. Again, the secret key is the concatenation of the secret keys for each one-time scheme. The public key is constructed using a hash tree T which is the complete binary tree with N leaves (we assume N is a power of 2). The nodes are labeled so that v_ϕ is the root node, v_0, v_1 are the left and right children, respectively, of v_ϕ, and $v_{\alpha 0}, v_{\alpha 1}$ are the left and right children, respectively, of v_α. Each node has contains a k-bit hash value where we denote by R_α the value stored in node v_α. The hash value in the j^{th} leaf (where we think of the j in binary representation) is $R_j = h(K_p^{(j)})$, where $K_p^{(j)}$ denotes the jth public key. For all non-leaf nodes, $R_\alpha = h(R_{\alpha 0}, R_{\alpha 1})$. The k-bit value R_ϕ stored in the root node is the public key for the N-time scheme.

When signing the rth message in the N-time scheme, the signer includes in the signature the usual one-time signature using the rth private key together with the corresponding public key and $R_{\alpha j} 1 \le j \le \log N$. The verifier checks that the signature is valid (assuming the public key sent is correct) and then verifies the key is valid by computing R_α for all α which are the prefixes of the binary representation of r. If the value computed for R_ϕ matches the public key, then the key was valid.

2.5.2.1 The LM Scheme

The LM scheme is a method designed to make more secure the conversion to N-time schemes using the hash tree, and also to make the messages shorter. To sign a long message, a single random k-bit string C is appended to the secret key. A corresponding string is appended to the private key $Z = h(C \,|I|n|\, 010)$, where I is the identity of the signer, n is meant to signify that the n^{th} message in an N-time scheme is being signed, and 010 (or any other 3 bit string) is a constant string.

To sign the message, the signer computes $h(M \,|C|I|N|\, 010)$ and signs using the n^{th} one-time scheme, as usual. In addition to sending the usual information in the signature, the signer also sends the random value C, with C being different for each pair I, n. The verification procedure proceeds as usual; however, the verifier must first compute $h(M \,|C|I|N|\, 010)$.

2.5.2.2 Guy Fawkes

Guy Fawkes is really a protocol designed for authenticating messages using hash functions and commitments to future keys. A commitment to the first key must be made known in a secure way. Then, subsequent keys are committed to in one message, to allow verification in the next.

This idea is appealing because only two hashes are involved at each step, the signatures are shorter than in the previous schemes, and an unlimited number of messages can be signed.

We should note that all messages must be received correctly because of the interdependence of the messages in the verification processes; before the $(n-1)^{st}$ message can be sent, the n^{th} message must be known (in order to commit to it). Furthermore, this protocol is prone to a man-in-the middle attack, provided an adversary can intercept two consecutive messages. From then on, the adversary can intercept all messages and replace them with his own. However, the algorithm can be modified so that an adversary must intercept 3, 4 or even N messages before beginning an impersonation at the expense of lengthening the size of the signature and making the messages more interdependent. We can prevent this attack by alternating this procedure with another signing method so that no N messages are signed the same way where N is the number of messages an adversary needs to intercept. This then renders the man-in-the middle attack not feasible.

The protocol: Let h be a hash function. Party A generates a series of passwords $X_0, X_1, X_2, \ldots$; A commits to X_j in message A_{j-1} and reveals it in message A_{j+1}. The commitment is

$$a_j = h(A_{j+1}, h(X_{j+1}), X_j)$$

The first message must be authenticated by some trusted means to start off the chain (e.g., RSA). Denote this trusted signature by sign (M).

First message: $(A_0, a_0, h(X_0), \text{sign}(A_0, h(X_0)))$

$(n-1)^{st}$ message: $(n>1)$: $(A_{n-1}, a_{n-1} = h(A_n, h(X_n), X_{n-1}), h(X_{n-1}), X_{n-2})$

n^{th} message: $(A_n, a_n, h(X_n), X_{n-1})$

In a_{n-1}, party A has committed to password X_n, the message A_n, and makes use of the current key X_{n-1}. In the n^{th} message, A reveals knowledge of password X_{n-1} as well as the hash of X_n so the verifier can check that $h(A_n, h(X_n), X_{n-1}) = a_{n-1}$. Furthermore, password X_{n+1} and message A_{n+1}, are now committed to in Step n, and so the process continues.

Suppose that an adversary wished to intercept messages and attempt to replace them with forged (man-in-the-middle) messages. Having intercepted $(A_n, a_n, h(X_n), X_{n-1})$, he cannot change the message, A_n, since a_{n-1} contains a commitment to it; he cannot change a_n, since it contains as input X_n, which has not been revealed yet but has been committed to in the previous message. Similarly, $h(X_n)$ was committed to in a_{n-1}, and he must also reveal the correct value of X_{n-1} in order to verify the against $h(X_{n-1})$ from the previous message. Since X_n is not known yet, but was committed to in the previous message, he must also send along the correct value of $h(X_n)$. Hence, if the adversary tries to forge any part of the message, it will not verify.

On the other hand, suppose an adversary has intercepted two messages:

N^{th} message: $(A_n, a_n = h(A_{n+1}, h(X_{n+1}), X_{n-1}), h(X_n), X_{n-1})$

N+1st message: $(A_{n+1}, a_{n+1}, h(X_{n+1}), X_n)$

The adversary now has information about two consecutive passwords and this gives him enough information to change parts of the n^{th} message (but not the intended message), and most of the $(n+1)^{st}$ message (including the intended message). The N and $(N+1)^{st}$ messages become:

Forged N^{th} message: $(A_n, a'_n = h(A'_{n+1}, h(X'_{n+1}), X_n), h(X_n), X_{n-1})$

Forged $N+1^{st}$ message: $(A'_{n+1}, a'_{n+1} = h(A'_{n+2}, h(X'_{n+2}), X'_{n+1}), h(X'_{n+1}), X_n)$ where a prime after a symbol denotes something changed by the adversary

After sending these two forged messages, the adversary has complete control of the system and can change all of the keys and messages from there on out. The adversary will only be caught if he stops spoofing the system. Note that the adversary needed two messages to get the value of X_n, and couldn't send along a forged message until the $(n+1)^{st}$, in order to make sure everything verified.

Modification 1

With a slight modification, it would be necessary for the adversary to intercept 3 messages before beginning to forge. In this case, the definition of a_n changes to:

$$a_n = h(A_{n+1}, h(X_{n+1}), h(X_{n+2}), X_n)$$

Three consecutive messages then look like:

$(n-1)^{st}$ message: $(A_{n-1}, a_{n-1} = h(A_n, h(X_n), h(X_{n+1}), X_{n-1}), h(X_{n-1}), X_{n-2})$

n^{th} message: $(A_n, a_n = h(A_{n+1}, h(X_{n+1}), h(X_{n+2}), X_n), h(X_n), X_{n-1})$

$(n+1)^{st}$ message: $(A_{n+1}, a_{n+1} = h(A_{n+2}, h(X_{n+2}), h(X_{n+3}), X_{n+1}), h(X_{n+1}), X_n)$

To verify the $(n+1)^{st}$ message, the user checks that $h(X_n)$ is correct in the n^{th} message, and that $a_{n-1} = h(A_n, h(X_n), h(X_{n+1}), X_{n-1})$ validates in the $(n-1)^{st}$ message. The addition of $h(X_{n+2})$ in the definition of a_n has the effect of making the validation process 3-message dependent, but it also serves the purpose of making it impossible (for similar reasons as above) to forge based on 2 consecutive intercepted messages. An adversary must intercept 3 consecutive messages in order to be able to begin forging messages. This process can be repeated to increase the message interdependency as well as the number of consecutive messages an adversary needs to intercept before he can begin forging.

Modification 2

One could also help to discourage a man-in-the middle attack for a scheme, where an adversary needs n consecutive messages before forging can begin, by encrypting every n^{th} message using a different method. If the adversary can not break this encryption scheme, he will never have enough information to successfully forge a verifiable message.

2.6 Polynomial Schemes

In recent years, asymmetric polynomial schemes have emerged as a possible alternative to asymmetric schemes based on the discrete logarithm or factoring problems. Although they have not been studied with as much intensity, they seem to offer desirable benefits such as smaller signature size, and as they are related to NP-hard problems, it seems reasonable that upon further analysis they will prove to be secure.

In 1988, Matsumoto and Imai [MI88] described the C^* Algorithm, which makes use of multivariate polynomials of degree two over a finite field. This scheme was subsequently broken by Patarin in 1995 [Pa95], and [Pa96] developed a new family of Asymmetric algorithms based on polynomials in which he modified C^* to avoid its weaknesses. This new family of algorithms is based on Hidden Field Equations (HFE). Its security relies on the difficulty of factoring a randomly selected system of multivariate quadratic equations over a finite field, which is an NP-hard problem.

2.6.1 Hidden Field Equations (HFE)

The HFE algorithms are described in [Pa96]. HFE signatures are generally of size 160 or 128 bits, but can be made as small as 64 or even 32 bits at the expense of more difficulty in verification. To understand HFE signatures, we must first explain HFE encryption.

The public information in HFE is the following:

1. A finite field K of $q = p^m$ elements (typically $p = 2$), and a length n
2. A set of n polynomials $(p_1,\ldots,p_n)$ in n variables over K
3. A method for putting formatting (redundancy) in messages

The secret items are:

1. An extension, L_n of K of degree n (where n is as in Item 1 above)
2. A function f from L_n to L_n with degree d (f is a polynomial, not too big, degree $d \leq 1024$, and generally, $17 \leq d \leq 64$)
3. Two affine bijections s and t from K^n to K^n

Once a basis for L_n/K has been chosen, an element of L_n can be represented by an n-tuple of elements from K: $(x_1,\ldots,x_n)$. The polynomial function f (secret item 2) is chosen in such a way that f can be represented by an n-tuple of polynomials of degree 2:

$$f(x_1,\ldots,x_n) = (p'_1(x_1,\ldots,x_n),\ldots,p'_n(x_1,\ldots,x_n)).$$

The polynomials in the public key (2) are a gotten from the composition of the functions t, f, and s (private key items 2 and 3): $t(f(s(x_1,\ldots,x_n))) = (p_1(x_1,\ldots,x_n),\ldots,p_n(x_1,\ldots,x_n))$. Since f can be

represented by the polynomials p' of degree 2, and s and t are affine functions, the composition results in the polynomial p, which is also quadratic.

2.6.2 HFE encryption and decryption

To encrypt a message $x = M$, redundancy is first added as prescribed in Item 3 of the public key (i.e., the way to add redundancy is common knowledge), and then the message is input into the public polynomials. The output is the encryption of the message:

$$HFE(X) = HFE(x_1,\ldots,x_n) = (p_1(x_1,\ldots,x_n),\ldots,p_n(x_1,\ldots,x_n)) = Y.$$

To decrypt, the public polynomials are 'inverted'. Since the polynomials are the result of the composition $t(f(s(x))$, we must invert t, s, and f. Since t and s are affine bijections, they can be easily inverted. To 'invert' f, we set $f = 0$ and solve for the roots of the resulting polynomial equation (this is the most difficult part of the algorithm). Since f is not necessarily invertible (it is a polynomial of degree d), we may get up to d solutions. We can identify the correct solution because of the redundancy added by the sender:

$$X = HFE^{-1}(Y)$$

2.6.3 HFE signature and verification

Let h be a public, collision-free hash function whose output is 128 bits (e.g., h is *MD5*). Let || denote the concatenation operation.

To sign a message:

1. Generate a small integer R with no block of bits of the form 10000 in its base 2 representation (e.g., start with $R = 0$)
2. Compute $h(R||10000||M)$
3. Consider the HFE Encryption Algorithm: $HFE(X) = Y$ where X and Y are 128 bits. With $Y = h(R||10000||M)$, try, using the secret key, to find a value X so that $HFE(X) = Y$. (Note that Y may have no inverse, since the secret polynomial f may not be invertible for that value. In that case, go back to 1 and try again by, for example, adding 1 to R). Once successful, the signature of M will be $R||x$

To verify a message:

1. Separate R and x from $R||x$ (x has a fixed number of bits so this is easy).
2. The signature is valid if $h(R||10000||M) = HFE(X)$.

2.6.4 Variation on Length of the signature

In the above signature, the length is the length of $R||x$, which will be just a few more than 128 bits. We can shorten the signature to about 64 bits as follows:

To sign a message:

1. Same as above
2. Same as above
3. Consider HFE Encryption Algorithm $HFE(X) = Y$ with X and Y of length 64 bits. Denote by F the public computation of HFE and F^{-1} one pre-image of HFE (if one exists) so that $Y = F(X)$.
4. Denote by h_1 the first 64 bits of the hash value $h(R||10000||M)$ and by h_2 the last 64 bits.
5. Compute $S = F^{-1}(h_1 \oplus F^{-1}(h_2 \oplus F^{-1}(h_1)))$, i.e., we search for at a value S such that $F(F(F(S) \oplus h_1) \oplus h_2) = h_1$.(**)

   If we are not successful (i.e., if HFE is not invertible for some step), then go back to Step 1 and try another R value). Otherwise, the signature of M is $R||S$.

To verify:

1. Separate R and S (S has a fixed length of 64), and then compute $h(R||10000||M) = h_1||h_2$
2. The signature is valid if the equation (**) above is satisfied

Comments on HFE:

- The length of the signature in this case is a little more than 64 bits, but notice that it requires three inversions of F in signing, which is more difficult than just 1 in the first scenario. In order to shorten the signature further, it is suggested to send only the first 32 bits of S. Although this is nice and small, it increases the verification process because the other 32 bits will have to be found by exhaustive search which is quite slow. Furthermore, we do not recommend a signature as small as 32 bits because it is too short to guarantee security.

- The polynomial f is specified so that the exponents of the variable are powers of q (the characteristic of K) or a sum of two powers of q (this is to guarantee that the public keys are quadratic polynomials). HFE recommends that the degree of f should be at least 17 and should have "enough" monomials of Hamming weight two in x. This is to avoid the "affine multiple attack." In general, it seems that to avoid this attack, it is enough to have at least one exponent with rather large Hamming weight. The HFE paper further suggests (based on intuition) that some of the coefficients of f should be general field elements of L as opposed to only using 0 or 1 (i.e., elements of K) as coefficients. Another attack to avoid is the "quadratic attack." The quadratic attack is possible if one can obtain lots of

quadratic relations on the plaintext. It is not immediately apparent if there is an easy check for this. Any f that is chosen should be analyzed to prevent against this or any other obvious attack.

- A computational concern arises about the time it will take to produce a signature that corresponds to the computation time for decrypting. The main work in the decryption lies in solving the equation $f(x)=a$.

- The size of the public key can get very large. In general, if $L=GF(2^N)$, then the size of the public key will be $N*(N(N-1))/2$ bits. The paper suggests a way to shorten the public key by keeping some of the public polynomials secret. This would shorten the key size but would also increase the signing side. Furthermore, there is no comment as to how this affects the security: It may or may not enhance it.

2.7 Cryptographic Primitives

Many of the Cryptographic algorithms above involve basic arithmetic in finite fields or rings. In fields (or rings) with large prime characteristic, often the most costly operation in the algorithm is modular exponentiation. In fields of characteristic two, elements sometimes get to be very long and difficult to deal with. We can often improve the efficiency of an algorithm by optimizing the field operations. We describe some such optimizations next.

2.7.1 Modular Multiple-Precision Exponentiation Algorithms

The process of computing a modular multiple-precision exponentiation (e.g., $r = g^x \bmod p$) is complicated and time-consuming because of the numerous multiple-precision modular multiplications involved. Public key data authentication algorithms that rely on exponentiation, in particular number theoretic schemes such as DSA and other El Gamal-type signature schemes, are burdened with a computationally complex operation. The Standard Square-and-Multiply Algorithm [K81] for computing an exponentiation involves $3*|x|/2$ multiplications on average, where $|x|$ is the number of bits in the exponent.

Several techniques exist for enhancing the performance of a modular multiple-precision exponentiation operation involving a fixed base and a random exponent or a random base and fixed exponent. For a fixed-base exponentiation, the best methods involve some form of pre-computation and storage of powers of the base. These techniques require a time (speed-up) versus space (amount of storage memory) trade-off. Techniques for random base exponentiation generally involve ways of representing the fixed exponent in nearly optimal ways such as in the use of addition chains.

The best technique for a given application depends on the type of exponentiation required (i.e., whether fixed-base or fixed-exponent), the security required, the amount of storage available, and the number of multiplications that performance allows. Gordon's publication [G98] of a survey of techniques for fast exponentiation is worth reading to determine the most suitable approach. In addition, the most recent literature should be searched as new enhancement

algorithms are continually proposed, such as the pre-computation technique by [BPV98], which is not in [G98].

2.7.2 Finite Field Arithmetic and Field Towers

The finite field

$$GF(2^m) \approx GF(2)[x]/f(x) = \{a_0 + a_1 x + ... + a_{m-1}x^{m-1} \bmod(f(x)) \mid a_i \in GF(2)\}$$

where $f(x)$ is an irreducible binary polynomial of degree m. An element $a \in GF(2^m)$ can therefore be represented as a m-tuple $a = (a_0, a_1, ..., a_{m-1})$ of 0s and 1s. Addition of two elements is a bitwise exclusive or:

$$a,b \in F \quad a+b = (a_1 \oplus b_1, a_2 \oplus b_2, ..., a_m \oplus b_m)$$

and multiplication is like a plain multiplication without any carries but with only the exclusive or accumulation. The result of the multiplication must, however, be reduced by the field polynomial $f(x)$. As the degree m of the field gets large, the multiplication can become time-consuming and the representation of the numbers can become big. For a general reference on finite field arithmetic, see [P1363].

If m is composite, we can use field towers to speed up the computations. Suppose $m = ns$. Then we can think of $GF(2^m) = GF(2^n)^s$ as a degree s extension of $GF(2^n)$. The elements are $a \in GF(2^m)$, $a = (\alpha_1, \alpha_2, ..., \alpha_s)$, where $\alpha_i \in GF(2^n)$.

For example, suppose $m = 156 = 12 * 13$. Then we can represent $GF(2^{156})$ as $GF(2^{13})^{12}$. The addition and multiplication of two elements $a = (\alpha_1, ..., \alpha_{12}), b = (\beta_1, ..., \beta_{12}), \alpha_i, \beta_j \in GF(2^{13})$ uses the underlying $GF(2^{13})$ arithmetic, which is much simpler than the arithmetic in $GF(2^{156})$. If m is highly composite, there are several possibilities for writing $GF(2^m)$ as an extension of a smaller field. We may even write $GF(2^m)$ as series of field extensions. For instance, $GF(2^{156}) = GF((2^3)^4)^{13}$, or $GF(2^{156}) = GF((2^2)^6)^{13}$, etc. We can speed up the arithmetic considerably in such fields. Thinking of $GF(2^{156})$ as a three-level tower would make the corresponding arithmetic about three times faster than the arithmetic in the field, using a polynomial basis.

2.8 Comparative Summary and Conclusions

The following table provides a comparative summary of the best viable candidate Public Key Authentication algorithms for the low-power environment, assuming parameter size recommendations specified in the sections describing each algorithm.

Table 2.7 Candidate Public Key Authentication Algorithms for the Low-Power Environment

Algorithm	Signature Length*	Storage Required*	Signature Operations	Verification Operations†††
Optimal ElGamal with pre-computation	40	109712**	$O(2n^2+2n)^{****}+O(h)^{\dagger}=$ $220+O(h)$, for $n=10$	$O(2(t+m-2)(2n^2+2n)))+O(h)^{\dagger\dagger}=$ $404544+O(h)$, for $t=80, m=8, n=48$
Storage Balanced ElGamal with pre-computation	40	36792**	$O(4n^2+4n)^{****}+O(h)^{\dagger}=$ $440+O(h)$, for $n=10$	$O(2(t+m-2)(2n^2+2n)))+O(h)^{\dagger\dagger}=$ $404544+O(h)$, for $t=80, m=8, n=48$
Feige-Fiat-Shamir	116	15456	$O(160n^2+160n)^{****}+O(h)^{\dagger}=$ $376320+O(h)$, for $n=48$	$O(161n^2+161n)^{****}+O(h)^{\dagger}=$ $378652+O(h)$, for $n=48$
ESIGN	96	192	$O(6n^2+6n)^{****}+O(h)^{\dagger}=$ $14112+O(h)$, for $n=48$	$O(6n^2+6n)^{****}+O(h)^{\dagger}=$ $14112+O(h)$, for $n=48$
Elliptic Curve El Gamal	40	120	$O(4n^2+4n)^{****}+O(h)^{\dagger}=$ $440+O(h)$, for $n=10$	$O(4n^2+4n)^{****}+O(h)^{\dagger}=$ $440+O(h)$, for $n=10$
Merkle Single-Bit	≤1088***	1985600**	Constant time	Constant time
Merkle Multiple-Bit	≤160***	36500**	Constant time	Constant time

*In bytes

**Assuming at least 5 (years) x 365 (messages/year) = 1825 messages to be signed

***Dependent on the length of the message. Assume a message length of 128 bits

****n is the number of single precision (16-bit) values used

†h is the number of 512-bit blocks in the message and assuming the use of SHA-1. Arithmetic computations generally outweigh the cost of hashing the message unless the message is long.

††Assuming that each exponent can be represented as $\sum_{i=0}^{t} e_i b_i, 0 \le e_i \le m$ and that we are using BGMW. Storage requirements should be adjusted to include the pre-computation table required by this algorithm.

†††Worst-case analysis.

We note that overall, the elliptic curve implementation of El Gamal seems to give the best performance without limiting the number of messages we can sign. We also recommend a closer look at Optimal El Gamal or N-time hash schemes if there is a number N such that, at most, N messages need to be signed. Furthermore, if messages aren't likely to be lost or skipped, we recommend a closer look at the Guy Fawkes hash-based scheme with the suggested modifications.

Deciding on an algorithm is only the first step in implementing a system for a low-power environment. Specific software and hardware optimizations must be carried out, and a wise choice of processors should be made for implementation.

In Section 7.3, we give a hardware design for the Elliptic Curve El Gamal signature scheme. The slowest part of the Signature Algorithm is computing a multiple of a point. We optimize the algorithm by using an innovative Point Halving Algorithm (patent pending) developed by [S00], one of the co-authors. The Point Halving Algorithm is more than three times as fast as point doubling in software, and we expect comparable performance in optimized hardware. Even after

the other parts of the overall point-multiplication process are accounted for, the performance gain from using point halving is roughly 2.5. We further optimize the algorithm by using field towers for the finite field arithmetic.

3. General Purpose Commercial Computing Platforms

All of the algorithms discussed in Section 2 must run on some computing platform. The algorithms must be implemented in hardware or in software. To fully realize any cryptographic functionality, a software implementation requires coding in some programming language, compilation into machine executable codes for a specific processor, and execution. For any given algorithm, the computing platform chosen for implementation has a significant impact on the performance as well as the power usage. This section of the report focuses on the general purpose computing platforms that lend themselves well to low-power public key cryptography applications. The computing platforms presented in this section are commercially available general-purpose processors and digital signal processors that offer a range of capabilities and are suitable for low-power public key cryptography. Section 5 presents custom processing platforms available either commercially or through in-house development.

3.1 Issues of Concern

One of the most important power-related factors associated with processor hardware is the power supply voltage necessary for operation. Traditionally, microprocessors have required 5 volts to power the processing device. Now, however, any low-power device almost certainly runs on a 3.3-volt power supply and some run on 2.7 volts. A 3.3-volt design generally costs more, but it does reduce power consumption by 34% (46% for 2.7-volt devices) compared to a 5-volt part, assuming the same current draw (normally a good assumption). Another processor feature found in low-power designs is power management functionality. A good candidate for low-power applications will have multiple user-controllable power regimes such as normal operation, idle mode where some basic functionality is still available, and power-down mode where only minimum functionality is available to reawake the processor. Each of these has decreasing power dissipation.

Beyond the electrical aspects of processors that decrease power dissipation, there are features that offer the potential for reducing the implementation complexity of a particular algorithm. These features reduce the time necessary to complete a computation, thus reducing total power consumption. The classic example is the use of digital signal processors for a variety of signal-processing tasks such as digital filtering and Fourier analysis. These processors have special data buses and memory which are separate from the program, and special instructions to enhance the performance of commonly executed functions. The following attributes of DSP chips offer the potential for fast modular multiple-precision multiplication, an important operation in most number theoretic algorithms for public key cryptography.

1. *Extended accumulator to sum multiple products.*

This feature is especially useful when the accumulator is large enough to sufficiently accommodate the overflow that occurs when performing multiple-precision multiplication using the convolution-sum method. For example, when two 128-digit numbers are multiplied using this method, the accumulator must be able to accommodate the overflow from 128 multiply-and-accumulate cycles. At most, this operation would require 7 or $\log_2(128)$ extra accumulator bits beyond the length of the multiplier and multiplicand.

2. *A multiply-and-accumulate (MAC) instruction.*

   The availability of a MAC instruction can result in extremely tight and therefore efficient code, especially when utilized in the multiplication of multiple-precision integers using the convolution-sum method. Since most cryptographic applications involve unsigned or non-negative integers, an unsigned-MAC instruction is desired. If a processor only supports a signed-MAC instruction, then each digit of multiple-precision integers must be reduced by one bit (the sign-bit).

3. *Parallel data buses.*

   Parallel data buses, available on many DSP chips, allow one to access data from multiple sources in the same instruction cycle. In addition, many processors allow data moves during an arithmetic operation.

4. *Multiple memory architectures (e.g., 1 for program memory and 2 for data memory).*

   The following code utilizes all four of the noted DSP attributes. The instructions are part of the inner loop of the Montgomery modular multiplication routine using the Motorola DSP56000.

```
mac x0, y0, a    x: (r0)–, x0    y: (r5)+, y0
mac x0, y0, a    x: (r1)–, x0    y: (r4)+, y0
```

   This example shows multiplication between the values in registers $x0$ and $y0$, accumulating their product into an extended bit accumulator register, a. During the same instruction cycle, registers $x0$ and $y0$ are loaded with values from memory in preparation for the next multiply and accumulate instruction.

   The efficient use of an extended accumulator with the use of a MAC instruction is not without some risk. For example, using a MAC instruction with an extended 36-bit accumulation register, the extra 4 bits in the accumulation register will accommodate 16 16-bit multiplies before overflowing. Therefore, some contingencies are necessary to support multiplying numbers greater than 256 bits. In addition, the MAC instruction often only supports signed arithmetic, making the processor less effective for cryptography.

   Another feature of some processors that may offer utility to cryptographic applications is the addition of a set of shadow registers to enable rapid context switching. In a cryptographic application, this may be important if multiple algorithms and/or multiple key sets are utilized in the same system.

In summary, the electrical specifications, power management functionality, architecture, and instruction set of commercially available processors is important to analyze for evaluation of their appropriateness in low-power applications. Section 3.2 evaluates some select microprocessors and microcontrollers, while Section 3.3 presents a survey of selected digital signal processors. All the processors included in these sections are attractive candidates for low-power public key cryptography for one reason or another. Section 3.4 provides a comparative summary of all the reviewed processors. See [L97] for a comprehensive microprocessor/microcontroller directory.

3.2 Microprocessors/Microcontrollers

Microprocessors are available commercially in 8-bit, 16-bit, 32-bit, and even 64-bit word sizes. Microcontrollers consist of a microprocessor and auxiliary functionality such as serial/parallel communications, timers, and interrupt controllers, all on the same chip, and are available in multiple word sizes as well. The smaller the word size for a given microelectronic technology, the smaller the device and the less power consumed. However, there is a trade-off in the computing capabilities offered in the different processors. For example, some 8-bit devices do not support a multiply or divide instruction.

Many microprocessors and microcontrollers offer some degree of power management, varying from software controlled and/or interrupt controlled idle modes to complete shutdown/sleep modes. The following subsections provide detailed information regarding specific processors identified as realistic candidates for low-power cryptography. All of the microprocessors and microcontrollers evaluated except the Toshiba/Motorola Echelon Neuron Chip include a multiply instruction in their set of instructions.

3.2.1 Hitachi H8 Microcontrollers

The Hitachi H8 series includes both 8-bit and 16-bit microcontrollers. The H8S is a 16-bit microcontroller with a 32-bit accumulator. The H8S/2655 series includes a MAC instruction. The Hitachi H8/3102 is a single chip microcomputer designed for use in smart card applications. This processor contains desirable features used to implement public key cryptography.

The H8/3102 8-bit CPU is designed with Hitachi's H8/300 CPU core. The H8/300 contains a speed-oriented architecture. On-chip memory includes 512 Bytes of RAM, 16 Kbytes of user ROM, and 8 Kbytes of EEPROM. The device operates on internal frequencies up to 5 MHz, has a sleep mode for power saving, and implements security features in the ROM and EEPROM. Hitachi offers the H8/3102 as both 5-volt and 3-volt devices.

The internal processor contains an instruction set that allows for ease in implementing Public Key algorithms. Features include arithmetic addition, subtraction, multiplication, and division. Most operations are computed on 8-bit or 16-bit boundaries.

www.hitachi.com

3.2.2 NEC V850/SA1 Microcontroller

The NEC V805 is a microcontroller with an integrated 16-bit RISC processing core, RAM, ROM, and flash options. It can perform a 16 x 16-bit multiply and includes multiply-accumulate and multiply-and-subtract instructions. Power management features include a halt mode (CPU clock stops, but peripherals continue to function), idle mode (CPU and internal-system clocks stop), and stop mode (everything stops, but register and memory contents stay intact).

www.nec.com

3.2.3 Toshiba/Motorola Echelon Neuron Chip

The Echelon Neuron Chip is a network communication control device that implements an encryption-based challenge response protocol for data authentication and is used across a wide variety of platforms. There are two major versions of the device: the 3120 and the 3150. The 3120 contains on-board ROM while the 3150 provides a medium for accessing off-board ROM. The 3120 and 3150 are manufactured by both Toshiba and Motorola.

The Neuron Chip contains three identical 8-bit CPUs. The first, Media Access Control, handles layers one and two of the 7-layer OSI model. The second CPU, Network Control, implements layers three through six of the OSI as well as the authentication algorithm. The third, Application CPU, is the user interface processor where user application code resides. The three-CPU architecture allows multi-tasking, pipelining, and parallel processing to occur, which enhances the performance of the device. The 3120 on-chip memory includes up to 2048 Bytes of RAM, 10 Kbytes of user ROM, and up to 2048 Bytes of EEPROM. The Neuron 3150 Chip includes 2048 bytes of RAM, 512 bytes of EEPROM, and an interface to address up to 43 Kbytes of external memory for application program and data use. The device operates on a selectable frequency range of 625 kHz to 10 MHz and has a sleep mode for power saving. The device is also ISO compatible.

While the native instruction set of the CPUs contains only very basic arithmetic functions such as add, increment, rotate, and shift, the Neuron C Compiler provides a library of functions built into the chip system image and allows for the ease of public key algorithm implementation. Functions include 16-bit unsigned integer multiply and divide, and 32-bit signed multiply, divide, addition, and subtraction.

3.3 Digital Signal Processors

Digital signal processing (DSP) chips are especially well suited to high performance implementations of exponentiation-based algorithms [DK90]. Montgomery multiplication, for example, is the basis for the Montgomery exponentiation operation. Montgomery multiplication can be written in such a way that it resembles a convolution operation, which is exactly what DSPs are designed to do. Due to their attractiveness in telephony and wireless applications, many DSPs are designed with low-power environments in mind. The following DSP chips offer the potential for fast low-power public key cryptography.

3.3.1 Analog Devices ADSP-2103

The ADSP-2103 is a 16-bit, fixed-point processor that can achieve a speed of 10.25 MHz using a 3.3-volt power source. Three separate arithmetic execution units comprise the data path: a MAC, an ALU (arithmetic/logic unit), and a barrel shifter which shifts 16-bit input to a 32-bit register. Although only one unit can be active during a single cycle, each is capable of single-cycle execution. The ALU provides operations such as increment/decrement, add-with-carry, and absolute value functions, as well as the standard ALU operations.

www.bdti.com/procsum/adi21xx.htm

3.3.2 Hitachi SH-DSP

The Hitachi SH-DSP is a general-purpose processor that is a combination of a 16-bit DSP and a 32-bit RISC microcontroller. It processes a single stream of instructions, and the DSP unit uses the microcontroller data path in combination with its own data path. The DSP unit contains a 16-bit fixed-point data path, a 40-bit ALU with eight guard bits, two 40-bit accumulators, six 32-bit operand registers, and a barrel shifter. The data path of the microcontroller uses sixteen 32-bit registers, some of which are used as address registers by the DSP. The Hitachi SH-DSP is compatible with Hitachi's popular SH-1 microcontroller. When used together, the functions and capabilities of the SH-DSP are similar to those of standard DPSs, and the SH-2 acquires several DSP-oriented features.

http://www.bdti.com/procsum/shdsp.htm

3.3.3 Lucent DSP1611/17

The Lucent DSP1611 and DSP1617 are high-end members of the DSP16xx family. Both are 16-bit fixed-point processors with two 36-bit accumulators and four guard bits for overflow. They use a 16-bit x 16-bit multiplier and a 32-bit ALU/shifter, both of which are capable of single-cycle execution.

www.bdti.com/procsum/dsp16xx.htm

3.3.4 Motorola DSP56L811

The Motorola DSP568xx family of processors offers microcontroller functionality with DSP instructions and architecture for parallel instruction processing. It is a 16-bit fixed-point processor with a 36-bit accumulator, thus allowing a maximum overflow of 4 bits. The DSP56L811 is a 2.7-volt part, and at 40 MHz draws a maximum of 20 mA.

http://www.bdti.com/procsum/dsp568xx.htm

3.3.5 NEC uPD7701x

The processors in this family have a 16-bit fixed-point architecture, and include both RAM- and ROM-based models. The data path consists of a 40-bit MAC, a 40-bit ALU, and a 40-bit barrel shifter, only one of which can be active at a time. Each is capable of single-cycle execution. The uPD7701x processors utilize a modified Harvard architecture with a program memory space and two additional data memory spaces. The uPD7701x does not provide a timer.

www.bdti.com/procsum/nec7701x.htm

3.3.6 TI TMS320LC5x

TI TMS320LC5x is a family of 16-bit fixed-point processors whose low-power members can run at a speed of 40 to 50 MIPS at 3.3 volts. The data path includes a 32-bit multiplier (capable of single-cycle execution), a 32-bit accumulator and secondary accumulator, a 32-bit ALU, and multiple barrel shifters. Separate from the fixed-point data path is an additional unit called the parallel logic unit, which allows additional operations.

www.bdti.com/proscum/ti320c5x.htm

3.3.7 Zilog Z89462

The Zilog Z89462 runs at 20 MIPS using a 3.3-volt power supply. It is the first member of the Z894xx family, and it is considerably more powerful than its Zilog predecessors. The data path includes a 16 x 16 ≥ 32-bit multiplier, a 32-bit ALU, and a barrel shifter. The Zilog Z89462 has on-chip memory as well as two off-chip memory interfaces, and an indexed addressing mode. Zilog plans to make microprocessors based on a derivative of the Z89462.

www.bdti.com/procsum/z89462.htm

3.4 Comparative Summary

The following table summarizes some salient features of the evaluated processors and provides a means to comparatively analyze competing computing platforms. However, the relevance of this information will quickly become obsolete due to the incredible pace at which advances in hardware occur.

Table 3.4 Comparative Features of General Purpose Computing Platforms

Device	Type	Word/ Acc size	Special Functions	Pwr (V)	Run Pwr (max)	Run Pwr (typ)	Sleep Pwr (typ)	Speed (MHz)
Hitachi H8	Cntrl	8 &16 bit	MAC	5		35mW	500μW	5
Hitachi H8S/2655	Cntrl	8 &16 bit	Signed MAC	3	186mW*	75mW*	54mW *	10

Device	Type	Word/ Acc size	Special Functions	Pwr (V)	Run Pwr (max)	Run Pwr (typ)	Sleep Pwr (typ)	Speed (MHz)
NEC V850/SA1	Cntrl	16 bit	MAC	3	-	30mW	16μW*	10
Echelon Neuron 31x0	Cntrl	3 8 bit	Machine code mult	5	-	-	-	10
AD ADSP-2103	DSP	16 bit	Unsigned MAC	3.3	46mW	28mW	-	10.24
Hitachi SH-DSP	DSP	16/40 bit	DSP/Microcontroller	3.3	-	660mW	-	60
Lucent DSP1611/17	DSP	16/36 bit	No rotate	2.7	-	86.4mW	-	50
Motorola DSP56L811	DSP	16/36 bit		3	90mW	60mW	6μW	40
NEC uPD7701x	DSP	16/40 bit	MAC, no timer	3.3	132mW	-	-	33
TI TMS320LC5x	DSP	16/32 bit	MPYU, MACD	3.3	-	86mW	-	40
Zilog Z89462	DSP	16/40 bit		3.3	99mW	-	-	20

4. Memory Storage

Memory is a vital part of any hardware system. It is especially useful for this application in both the storage of pre-computed values and/or interacting with a processor.

This document suggests that the storage of pre-computed values may be a desirable feature in implementing public key algorithms if the power savings is sufficient to warrant external memories. Additionally, when working with processor units, external memories are necessary for holding processor instructions and data for interacting with other circuits.

Small memories can be implemented directly on the custom-designed integrated circuit. It is usually not feasible to implement large memory arrays into a single custom device. Several vendors now provide low-power options for large memory arrays as shown in the following sections. The following sections also discuss memory options for implementing memories on custom integrated circuits and as stand-alone devices. The intent of this section is to present memory options, not to recommend a specific memory device or implementation method. Furthermore, memory devices continue to advance rapidly, and many of the devices specified in the tables below could quickly become obsolete. All memories discussed in this section contain information on active power consumption where the memories are being accessed during a write or read cycle. In addition, stand-by power is also reported in which the chip-select lines hold the device in an inactive low-power state.

4.1 Memory Storage on Custom Designed Circuits

Several types of memories can be implemented on a custom integrated circuit. However, the two basic types of memories are Random Access Memories (RAMs) and Read-Only Memories (ROMs). The following table shows a comparative study of ROM memories on board a single integrated circuit device. The table is provided only as an example of what is possible and what the potential power consumption may be, because custom circuit characterization is very

dependent on its technology. If a custom circuit design is chosen, once the target technology is selected, a thorough power analysis for that technology should be performed.

Table 4.1 Comparative Study of ROM Memories

Foundry	ROM Size	Standby Power Dissipation (uW)	Dynamic Power Dissipation (mW/MHz)		Dynamic Power Dissipation at Frequency (mW)	
			5.0 Volts	3.3 Volts	5.0 Volts	3.3 Volts
Sandia-MDL (0.6 micron)	2K x 8	0	1.3927	0.8390	11 mW@ 8MHz 16 mW @12Mhz	6.7 mW@8MHz 10 mW@12MHz
Sandia MDL (0.6 micron)	2K x 16	0	2	1.21	16 mW@ 8MHz 24 mW @12Mhz	9.7 mW@ 8MHz 14 mW @12Mhz
Honeywell (0.8 micron)	2K x 8	0			128 mW@ 8MHz 192 mW @12Mhz	

4.2 Read Only Memories

Read Only Memories (ROMs) allow data to be read from the device. ROM is basically a chip that contains information that can be "looked up" or addressed to retrieve information stored on the device.

There are several types of read-only memories. Some have information stored on them during manufacturing. These are referred to as mask-programmed read-only memories. Others can be programmed or loaded with information by the user and are referred to as Programmable Read-Only Memories (PROMs). There also are memories that can be erased by ultra-violet light and re-written using special equipment or special programming techniques. These are referred to as Erasable Programmable Read-only Memories (EPROMs).

This memory evaluation focuses on Erasable Programmable Read-Only Memories. The two most common types of ERPOM are traditional EPROM and FLASH. Flash memories tend to have a lower operating voltage and much larger capacity.

4.2.1 Atmel

Atmel offers a large selection of EPROMs and Flash memories with 5-volt, 3.3-volt, and 3.0-volt components. The EPROM minimum access times for the components range from 45ns to 150ns. The Flash memory access times range from 120ns to 350ns. They offer several different configurations and memory sizes.

www.atmel.com

4.2.2 Advance Micro Devices (AMD)

AMD offers EPROMs for the 5-volt technology and Flash memories in 5-volt, 3.3-volt, and 1.8-volt. The EPROM access times range from 45ns to 250ns. Flash access times range from 55ns to 120ns. They offer several different configurations and memory sizes.

www.amd.com

4.2.3 Cypress

Cypress only offers EPROMs for the 5-volt technology at this time. The minimum access times for the components range from 25ns to 200ns. They offer several different configurations and memory sizes.

www.cypress.com

4.2.4 Intel

Intel offers Flash memories in both the 5-volt and 3.3-volt technologies. They offer several different configurations and memory sizes.

www.intel.com

4.2.5 Samsung

Samsung offers Flash memories in both the 5-volt and 3.3-volt technologies. Flash access times for their 4MB device is 120ns. They offer several different configurations and memory sizes.

www.samsung.com

4.2.6 Micron

Micron offers Flash memories in both the 5-volt and 3.3-volt technologies. Flash access times for their 4MB devices is range from 60-100ns. They offer several different configurations and memory sizes.

www.micron.com

4.2.7 EPROM Comparative Study

Below is a table representing the EPROM compiled data. All memory power consumption calculations are based on accessing the memories at a slower rate than their true access speed.

Table 4.2 EPROM Data Compilation

Vendor	Part Number	Org.	Nominal Operating Voltage (volts)	Access Speed (ns)	Power Consumption			
					Active Typ. Power (mW)	Active Max. Power (mW)	Stand-By Typ. Power (uW)	Stand-By Max. Power (uW)
Atmel	AT27C256R	32Kx8	5	45-150		100[1]		500
	AT27LV256A		3.3	55-150		27[1]		66
	AT27BV256		3.0	70-150		24[1]		60
	AT27C010(L)	128Kx8	5	45-150		125[1]		500
	AT27LV010A		3.3	70-150		27[1]		66
	AT27BV010		3.0	90-150		24[1]		60
AMD	AM27C256	32Kx8	5	45-250		125[2]		500
	AM27C010	128Kx8	5	45-250		150[2]		500
Cypress	CY7C271A	32Kx8	5	25-55		275[2]		75(mW)
	CY7C010	128Kx8	5	45-200		250[1]		75(mW)

[1]Active power based on 200ns cycle time
[2]Active power based on 100ns cycle time

4.2.8 Flash Memory Comparative Study

Below is a table representing the Flash memory compiled data. All memory power consumption calculations are based on their slower speed version. Power consumption can be reduced by accessing the devices at a slower rate. The devices listed in the table below are all 4MB devices.

Table 4.3 Flash Memory Data Compilation

Vendor	Part Number	Org.	Nominal Operating Voltage (volts)	Access Speed (ns)	Power Consumption			
					Active Typ. Power (mW)	Active Max. Power (mW)	Stand-By Typ. Power (uW)	Stand-By Max. Power (uW)
Atmel	AT29C040A	512KX8	5	120-200		200		500
	AT29LV040A		3.3	200-250		50		132
	AT29BV040		3.0	250-350		45		120
AMD	AM29F004B	512KX8	5	55-120	100	150	2(mW)	5(mW)
	AM29LV040B		3	70-200	21	36	.6	15
Intel	28F004B5	512KX8	5			300		100
	28F004B3		3.3		33	60	24	66
Samsung	KM29W040AT	512KX8	5	120	50	100	50	275
			3.3	120-200	17	33	33	165
Micron	MT28F004B5	512KX8	5	60-80		250		650
	MT28F004B3		3.3	90-100		50		330

4.3 Random Access Memories

Random Access Memories (RAMs) allow data to be written in and read out. These memories are widely used when interfacing with a processor or when data needs to be saved in intermediate states. In a system, data is written to the memory and then read from memory as often as necessary.

There are also several types of RAM; however, this report focuses on 1MB low-power asynchronous Static Random Access Memories (SRAMs). As shown below, several vendors now offer low-power SRAMs.

4.3.1 Hitachi

Hitachi offers 5-volt, 3.3-volt, and 3.0-volt components in the low-power arena. The minimum access times for the components range from 55ns to 70ns. They are configured as 128K x 8 memories. They offer a 64K x 16-bit arrangement for 1MB memories, as well as smaller and larger memory sizes.

www.hitachi.com

4.3.2 Cypress

Cypress offers 5-volt, 3.0-volt, 2.6-volt and 1.8-volt components for low-power applications. The minimum access times for the components range from 55ns to 200ns. They are configured as 128K x 8 memories. They offer a 64K x 16-bit arrangement for 1MB memories, as well as smaller and larger memory sizes.

www.cypress.com

4.3.3 Samsung

Samsung offers 5-volt, 3.0-volt, 2.5-volt and 1.8-volt components for low-power applications. The minimum access times for the components range from 55ns to 300ns. They are configured as 128K x 8 memories. They offer a 64K x 16-bit arrangement for 1MB memories, as well as smaller and larger memory sizes.

www.samsungsemi.com

4.3.4 Mitsubishi

Mitsubishi offers 5-volt and 3.3-volt components for low-power applications. They also offer 2 versions of 3.0-volt components. The minimum access times for the components range from 55ns to 150ns. They are configured as 128K x 8 memories. They offer a 64K x 16-bit arrangement for 1MB memories, as well as smaller and larger memory sizes.

www.mitsubishichips.com

4.3.5 Performance Semiconductor

Performance Semiconductor offers 5-volt and 3.3-volt components for high speed and low-power applications. The minimum access times for the components range from 15ns to 70ns. They are configured as 128K x 8 memories. They offer smaller memory sizes as well, in several configurations.

www.performance-semi.com

4.3.6 GSI Technology

GSI Technology offers 3.3-volt components for high speed and low-power applications. The minimum access times for the components range from 8ns to 15ns. They are configured as 128K x 8-bitmemories. They offer both smaller and larger memory sizes, as well as several configurations.

www.gsitechnology.com

4.3.7 Comparative Study

The SRAM data is compiled in the table below. All memory power consumption calculations are based on their slower speed version. For example, if a vendor offers both a 55ns and 70ns version of a component, power consumption calculations are based on the 70ns version. Additionally, many of these parts can be accessed slower than the specified range, thus reducing power consumption even more.

Table 4.4 SRAM Data Compilation

Vendor	Part Number	Org.	Nominal Operating Voltage (volts)	Access Speed (ns)	Power Consumption			
					Active Typ. Power (mW)	Active Max. Power (mW)	Stand-By Typ. Power (uW)	Stand-By Max. Power (uW)
Hitachi	HM628128D	128K X 8	5	55-70		300	5	100
	HM62W8128D		3.3	55-70		132	3.3	66
	HM62V8128D		3.0	70-85		90	2.4	60
Cypress	CY62128	128K X 8	5	55-70	150	300	2	100
	CY62128V		3.0	70	60	120	1.2	45
			2.6	100	45	60	0.9	30
			1.8	200	30	45	0.9	30
Samsung	KM681000C	128K X 8	5	55-70	225	300	1.5	50
	KM68FV1000		3.0	70-85		120		15
	KM68FS1000		2.5	120-150		78		13
	KM68FR1000		1.8	300				9

Vendor	Part Number	Org.	Nominal Operating Voltage (volts)	Access Speed (ns)	Power Consumption			
					Active Typ. Power (mW)	Active Max. Power (mW)	Stand-By Typ. Power (uW)	Stand-By Max. Power (uW)
Mitsubishi	M5M51008	128K X 8	5	55-100	175	350		100
			3.3	70-100	66	115		40
			3.0	120-150	45	90		33
	M5M5V108		3.0	70-100		90		14.4
Performance Semiconductor	P4C1024L	128K X 8	5	55-70		350		100
	P3C1024		3.3	15-35				
GSI Technology	GS71108	128K X 8	3.3	8-15		231		33(mW)

5. Special Purpose Computing Platforms

Special purpose computing components specifically designed to address secure applications are available commercially. Some of these components were designed for smart card applications and others specifically for use in cryptography applications. This section will summarize some of these components.

Many of the devices listed below are subject to import/export control laws. Depending on the application, there may be some import/export compliance laws that must be adhered to before implementing the device.

5.1 SGS Thompson ST16CF54

The SGS Thompson ST16CF54 is a serial access microcontroller specially designed for smart card applications. This processor can be used to directly implement Public Key algorithms.

The ST16CF54 has an internal 8-bit modular arithmetic processor that is designed to speed up cryptographic calculations required in public key algorithms. On-chip memory includes 512 bytes of RAM, 16K of user ROM, and 4K of EEPROM (electrically erasable PROM). The device operates on internal frequencies of up to 5 MHz, has a sleep mode for power saving, and implements security features. The device is also serial access ISO compatible and contains a number generator. SGS Thompson offers the ST16CF54 as a 5-volt device.

The modular arithmetic processor contains a library of firmware functions in its system ROM that allows for the ease in implementing public key algorithms. Some of the firmware functions include calculating Montgomery constants for appropriate mathematical implementation of modular calculations, basic mathematics for modular and non-modular operations on operands up to 1024 bits, modular exponentiation on operands up to 1024 bits, and functions such as RSA-based operations, digital signatures, and hashing algorithms.

5.2 Motorola MSC0501

The Motorola MSC0501 is a serial access microcontroller specifically designed for use in embedded conditional access systems and other security conscious systems. This processor can be used to directly implement public key algorithms.

The MSC0501 8-bit CPU is based on the industry-standard M68HC05 with additional hardware to make it a modular arithmetic processor. On-chip memory includes 896 bytes of RAM, 20K of user ROM, and 4K of EEPROM. The device operates on internal frequencies up to 5 MHz, has wait and sleep modes for power saving, and implements security features. The device is also serial access ISO compatible and contains a custom random number generator. Motorola offers the MSC0501 as both a 5-volt and 3-volt device.

The modular arithmetic processor allows the MSC0501 to handle complex mathematical calculations required in many applications using public key cryptography for authentication and signature verification. The modular arithmetic processor is a Montgomery engine and has been designed to perform Montgomery calculations such as Montgomery modular multiplication and Montgomery modular squaring. The hardware allows modular arithmetic to be performed on numbers up to 1024 bits in length. The modular arithmetic processor can be accessed directly by user software.

5.3 Siemens SLE44CR80S

The Siemens SLE44CR80S is a secure microcontroller specifically designed for smart card applications. This processor can be used to directly implement public key algorithms.

The SLE44CR80S 8-bit CPU is designed to work hand-in-hand with a 540-bit arithmetic co-processor. On-chip memory includes 606 Bytes of RAM, 16K of user ROM, and 8K of EEPROM. The device operates on internal frequencies up to 5 MHz, has a sleep mode for power saving, and implements security features. Siemens offers the SLE44CR80S as both a 5-volt and 3-volt device.

The arithmetic co-processor allows the SLE44CR80S to handle complex mathematical calculations required in public key applications. The co-processor performs exponentiation modulo operations on 512 bit operands.

5.4 Philips P83C858

The Philips P83C858 is a serial microcontroller specifically designed for smart card applications. This processor can be used to directly implement public key algorithms.

The P83C858 8-bit CPU core is based on the industry standard 8051 with a Fast Accelerator for Modular Exponentiation (FAME) co-processor for use with public key algorithms. On-chip memory includes 640 bytes of RAM, 20K of user ROM, and 8K of EEPROM. The device operates on frequencies up to 8 MHz and implements security features. The device is also serial access ISO compatible. Philips offers the P83C858 as a 5-volt device.

The FAME co-processor allows the P83C858 to handle complex mathematical calculations required in public key applications. The co-processor performs key generation up to 2048 bits and performs calculations such as 1024-bit RSA signature processing.

5.5 Co-processor for Cryptography Applications

The Co-processor for Cryptography Applications is an Application Specific Integrated Circuit (ASIC) that has been developed at the Technical University of Madrid. The device was specifically designed for a customer under a Non-Disclosure Agreement. The only publication/documentation on this device is [RML97]. The device was designed specifically for cryptography applications.

The ASIC was designed to work as a co-processor for a generic CPU. Internal memory consists of 1024 bytes of data. The co-processor's maximum speed is 33 MHz. We have no knowledge of sleep mode or security features.

The co-processor supports a set of instructions that allow for the ease of public key algorithm implementation. Desirable instructions include performing Montgomery multiplication on 762-bit data and modular exponentiations with exponents greater than 64 bits.

5.6 Motorola Advanced INFOSEC Modules

Motorola's Advanced INFOSEC Module (AIM) is an integration of several devices used to create a system solution for cryptography applications.

The Advanced INFOSEC Module contains an AIM VLSI chip that is the heart of the cryptography system. It contains three RISC processor engines that can simultaneously process data. The Key Management Cryptographic Engine (KMCE) is the master controller in the AIM VLSI. It contains a ROM-based Secure Operating System running on a high performance, 32-bit RISC processor with a math co-processor designed for public key algorithm processing. The Programmable Cryptographic Processor (PCP) contains two high-speed engines (Programmable Crypto Engine, PCP and the Configurable Crypto Engine, CCE) developed to perform channel encryption and decryption, and data processing typically used in secure communications signaling. The Programmable Crypto Engine is optimized for processing Codebook-style algorithms and the Configurable Crypto engine is optimized for processing Combiner-style algorithms. The processors operate independently at 100 MHz (i.e., 4-stage pipelined). The VLSI AIM implements a power management system and realizes very high security features. The VLSI AIM device is a 3.3-volt device. The AIM VLSI is fully programmable.

The AIM VLSI implements a mod-N solution extractor (NSE) which is a special purpose co-processor for the acceleration of public key arithmetic.

The Advanced INFOSEC Module can be purchased as a full system with user interface, as a module to embed into a custom system, or as a chip to embed onto a custom design module.

6. Custom Design Computing Platform

A custom design platform offers the most flexibility in giving the customer exactly what is needed. Custom designs can address issues such as speed, power, security, and functionality. The entire computing platform can exist on a single Application Specific Integrated Circuit (ASIC) or can be integrated with commercially available components to produce the desired computing platform. Reconfigurable logic (Field Programmable Gate Arrays [FPGAs] and Programmable Logic Arrays [PLAs]) can be just as easily targeted as ASICs, depending on the needs of the design.

This section will discuss low-power design techniques for a custom-computing platform. In the next section, we will see actual implementation of algorithms that can be targeted to a custom-integrated circuit (ASIC, FPGA, PLA) for use in low-power public key cryptography applications.

6.1 General Design Techniques for Low-Power Applications

Designing Integrated Circuits (ICs) for a low-power environment is dependent upon three factors: Frequency, Voltage, and Capacitance (Power = $F*V^2*C$). The characteristics of these elements and how they affect the power dissipation of a CMOS (Complementary Metal Oxide Semiconductor) Integrated Circuit will be discussed in this section. A detailed report for designing low-power devices can be found in [CB95].

6.1.1 Frequency

Frequency is one element that impacts the power consumption of a CMOS Integrated Circuit. An IC can be designed in such a way to minimize the effect of the frequency and control the power dissipation.

One commonly used design technique often referred to as sleep mode is to turn the device's frequency (clock) off when not in use. Sleep mode dramatically reduces the power consumption since transistors throughout the device are no longer charging and discharging gate input capacitance. Likewise, this technique can be implemented on blocks of logic to reduce the power consumption during operations (partial sleep mode). For example, if operations in a device are sequential or periodic, the clock can be disabled for those operations when they are not in use, thus reducing the overall power consumption. A gated clock or input sensing circuitry are commonly used design techniques to implement sleep modes.

Similarly, the data bus structure can be designed to reduce the power consumption. Data buses typically route to several locations throughout a design, and charge and discharge several gate input capacitances. Thus, all receiving locations of the address bus may consume unnecessary power if it is not processing the data. For example, memory devices (ROM and RAM) typically share a common data bus. Most designs do not need to communicate with the devices concurrently, so power consumption may be reduced by addressing only the necessary target device while disabling the other data bus receivers.

Finally, the operating frequency of the device has a direct impact on the power consumption. Do not operate the device at a higher frequency than is necessary.

6.1.2 Voltage

The operating voltage is another factor contributing to power consumption. Basically, the lower the voltage, the lower the power consumption. If feasible, select a device that can operate at a low voltage.

Advanced design techniques adjust the logic swing of internal devices to be lower than the operating voltage, which reduces power demands.

6.1.3 Capacitance

As discussed in the frequency section, internal gate capacitance of the integrated circuits also affects the power consumption; however, gate capacitance is difficult to control since it is typically based on the design technology, which in most cases the designer has no direct control over. The technology refers to the feature size, such as 1 micron, 0.5 micron, etc., for a design process. The smaller the size of the technology feature, the less the capacitance and the less power consumed.

6.2 Design Techniques for Random Number Generation

A desirable feature for public key cryptography applications is the ability to generate keys. Depending on the needs, both pseudo random numbers and true random numbers can be generated on an Application Specific Integrated Circuit. The techniques discussed in this section can be modified to address the specific needs of the device.

6.2.1 Pseudo Random Number Generation

A linear feedback shift register (LFSR) is a pseudo random number generator that is easy to implement using digital logic. It provides bit sequences that can be generated over a long period of time. In theory, an n-bit LFSR can generate $2^n - 1$ bit-long pseudo random sequences before repeating.

A variation of the LFSR is the self-shrinking generator [MS94]. This implementation uses pairs of outputs from an LFSR to generate the output bits. This destroys most of the algebraic structure of the LFSR sequence, making it more difficult to analyze the sequence.

One-way functions such as DES or SHA can be used to generate pseudo random bit sequences by applying the function to a pseudo-random seed, and then applying the function to the sequence of values [MvV97, pp. 173–175].

6.2.2 True Random Number Generation

For ASIC devices, one of the best ways to generate true random numbers is to use random noise generated by the properties of the device. G.B Agnew [A88] proposed using metal insulator semiconductor capacitors. If two such capacitors are placed in close proximity, the random bit is a function of the difference in charge between the two. A similar method of producing random numbers is to sample the bits generated by the thermal noise of a resistor [T99]. Another random number generator generates a random bitstream based on the frequency instability in a free running oscillator [FMC84].

7. A Custom Design of Candidate Low-Power Algorithms

Taking into consideration low-power design techniques, VHDL (VHSIC Hardware Description Language) was used to capture the functionality of the public key cryptography operations and algorithms. Some of the best candidate algorithms, including Optimal El Gamal and El Gamal for elliptic curves, were also synthesized to hardware gates, and power analysis was performed. Our strategy was to implement a library of mathematical operations and then use them to build the algorithms. The key sizes used were 160 for Optimal El Gamal, and Elliptic Curve El Gamal was implemented for both 89- and 178-bit key sizes.

7.1 Design of Generic Mathematical Operations

We designed a generic set of mathematical operations for operand sizes of 160 bits that could be easily adapted to algorithms. We began with basic hardware designs such as a ripple adder. We also implemented versions of add, subtract, multiply and divide from the *Handbook of Applied Cryptography* [MvV97]. The following table summarizes the operations that were implemented and their respective gate count and speed for 160-bit operands.

Table 7.1 Size and Speed Comparisons of Implementations of Mathematical Operations

Mathematical Operation	Operand Size	Synthesized Gate Count[3]	Period (ns) (Clock speed)	Number of Clocks
Ripple Adder[1]	160	2361	75	1
HAC[2] Add –8	160	5216	20	40
HAC[2] Add –32	160	4912	25	10
Ripple Subtracter[1]	160	2442	80	1
HAC[2] Sub –8	160	5226	20	40
HAC[2] Sub –32	160	4940	25	10
Array Multiply	160 x 160	~179,000	~179,000	1
Shift/Add Multiply	160 x 160	6110	85	640
HAC[2] Mult –8	160 x 160	11041	25	40
HAC[2] Mult –32	160 x 160	19284	60	10
HAC[2] Mult –8 w/ 128 x 8 RAM	160 x 160	2300 0.6x0.5 mm	30	40

Mathematical Operation	Operand Size	Synthesized Gate Count[3]	Period (ns) (Clock speed)	Number of Clocks
Array Divide	320/160	~179,000	~179,000	1
Shift/Subtract Divide	320/160	7942	85	640
HAC[2] Div –8	320/160			
HAC[2] Div –32	320/160			
Software (8031, 2K x 8 RAM, 4K x 8 ROM)	Generic	~12K 1.8 x 1.6mm 1.1 x 0.91mm	~80	Thousands of Clocks

NOTES:

1) Ripple Adder and Subtracter is a ripple carry implementation.

2) Handbook of Applied Cryptography (HAC) implementations with an operand width of X bits. HAC operations were implemented with registers, not memories. Memory implementation would result in smaller (net) values.

3) Gate counts obtained from synthesizing VHDL with Synopsys Design_compiler tool (effort of medium) and targeting a 0.6-micron library.

7.2 Design of Optimal El Gamal Signature Algorithm with Pre-Computation

As described in Section 2.1.3, the Optimal El Gamal Signature Algorithm performs the following general computations per message:

$$r = (g^{k} \bmod p) \bmod q$$

$$s = (rxH(m) - k) \bmod q$$

Assuming pre-computation of r, k, and rx with $a = rx$, the computation becomes

$$s = (aH(m) - k) \bmod q$$

where $H(m)$ is the 160-bit hash of the message. a, k, and q are also 160 bits.

A Read Only Memory (ROM) is needed to store three (r, k, and rx) pre-computed 160-bit values per computation. The major operations for the above computation are

1. Hash of the message (implemented using SHA-1 Algorithm)
2. One Montgomery Multiply on two 160-bit values
3. A modular subtraction using two 160-bit values

The signature is composed of two 160-bit values (r and s), where r is read from memory and s is computed as described above.

7.2.1 Optimal El Gamal Hardware Implementation

The hardware implementation of Optimal El Gamal Algorithm was implemented using VHDL (see Figure 7.1). It assumes that *r, k,* and *rx* are pre-computed. The ROM interface that controls the reading of these pre-computed values was not modeled at this time. However, it can be easily added once a ROM is selected, with minimal impact to the overall size of the design.

The VHDL was synthesized to create the following block diagram showing the functional implementation. The target synthesis ASIC library was the CMOS6 library, which is 0.5um, 5-volt process (CMOS6R is a radiation-hardened version of CMOS6). Essentially, there are four major blocks: opt_cntrl, sha1, mont_mult, and mod_sub. The implementation is data-path *oriented*, which means that data flows from the input to output with very little looping.

For hardware implementation, the total number of compute cycles from start to finish is approximately 488 clocks. The slowest static timing analysis data path is 88ns. With these two numbers in mind, the algorithm as presently designed can be run at 10 Mhz (100ns cycles) and compute a 512-bit message signature result in (100ns x 488 clocks), approximately 49 microseconds (μs)

- *Number of Clock Cycles for Completion: 488*
- *CMOS6 Approximate Gate Equivalents: 32,577*
- *CMOS6 Slowest Timing Path: 87.76 (from Montgomery Multiplication)*
- *CMOS6R Approximate Gate Equivalents: 32,067*
- *CMOS6R Slowest Timing Path: 108.78 (from Montgomery Multiplication)*

The estimated power (as estimated by Synopsys Power Compiler tool) for a 0.5um, 5-volt technology is as follows:

- *CMOS6R Power Estimation @5V, 100ns: 44.042mW*
- *CMOS6R Power Estimation @5V, 200ns: 22.239 mW*

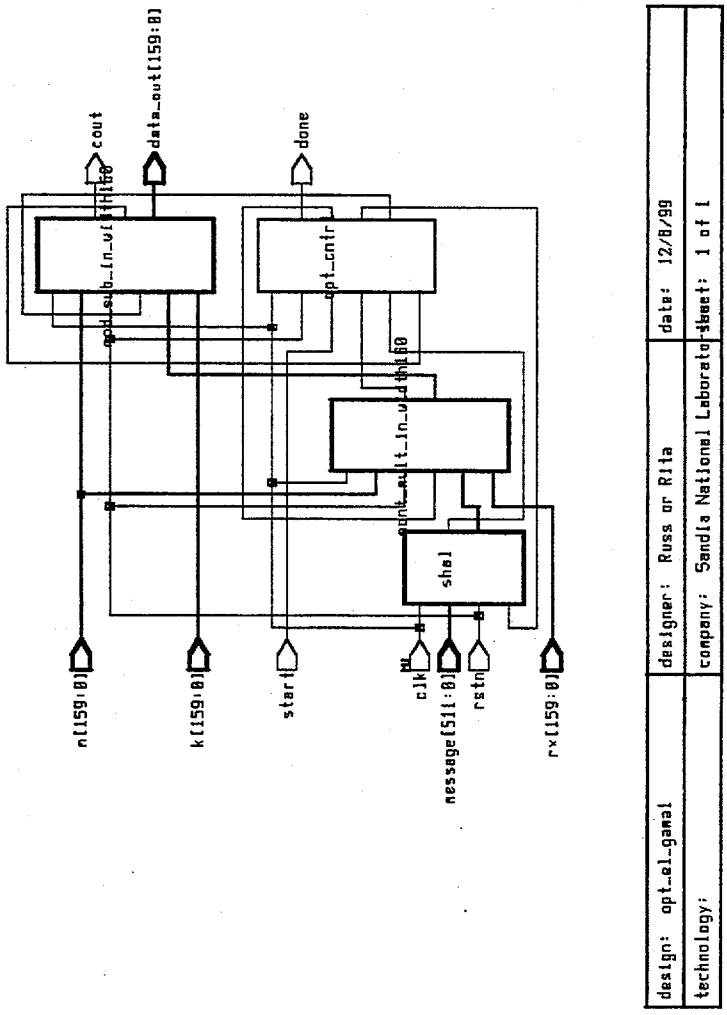

Figure 7.1 Optimal El Gamal Hardware Implementation Diagram

7.2.1.1 Optimal El Gamal Controller

Opt_cntrl is the control state machine that controls all the operations for the Optimal El Gamal Algorithm. It controls and determines when each block in the data path flow is to begin and end data manipulation. The controller is fast and is a very small portion of the overall design. It is implemented using state machine methodology. Statistics for our implementation are as follows:

- *CMOS6 Approximate Gate Equivalents: 78*
- *CMOS6 Slowest Data Timing Path: 2.14 ns (register to register)*
- CMOS6R Approximate Gate Equivalents: 74
- *CMOS6R Slowest Data Timing Path: 2.01 ns (register to register)*

7.2.1.2 Secure Hash Algorithm-1

SHA-1, the Secure Hash Algorithm, is the first data block in the design that performs the 1st set of data manipulation operations. SHA-1 was implemented per FIPS Standard (FIPS180-1). SHA-1 computes $H(m)$, the hash of the incoming message, and outputs a 160-bit message digest. SHA is the largest hardware block in the Optimal El Gamal Signature Algorithm. Statistics for implementation are as follows:

- *Number of Clock Cycles for Completion: 162*
- *CMOS6 Approximate Gate Equivalents: 19,744*
- *CMOS6 Slowest Timing Path: 35.82 ns (register to register)*
- *CMOS6R Approximate Gate Equivalents: 19,715*
- *CMOS6R Slowest Timing Path: 42.66 ns (register to register)*

7.2.1.3 Montgomery Multiplication

The Montgomery Multiplication block, Mont_mult, performs the Montgomery multiply between the 160-bit message digest, $H(m)$, and the 160-bit pre-computed rx stored in memory. This algorithm assumes that rx is in reduced Montgomery form. The output of this block is a 160-bit number. The implementation of the Montgomery Multiplication Algorithm is relatively small; however, the static timing analysis indicates this block contains the longest delay path.

- *Number of Clock Cycles for Completion: 324*
- *CMOS6 Approximate Gate Equivalents: 7,900*
- CMOS6 Slowest Timing Path: 87.76 (register to register)
- *CMOS6R Approximate Gate Equivalents: 7,592*
- *CMOS6R Slowest Timing Path: 108.78 (register to register)*

7.2.1.4 Modular Subtraction

The final step in the Optimal El Gamal Algorithm is a modular subtraction. The mod_sub block computes the modular subtraction of the 160-bit value k from the 160-bit value output by the Montgomery Multiplication Algorithm. The size is relatively small; however, the delay path is relatively long due to the size of the operands. If the design allows, this function may be eliminated by initializing the Montgomery multiplier with $q - k$ instead of zero, since k is subtracted from the output of the multiplication, thus eliminating the final modular subtraction.

- *Number of Clock Cycles for Completion: 4*
- *CMOS6 Approximate Gate Equivalents: 4,670*

- *CMOS6 Slowest Timing Path: 79.45 (input to register)*
- *CMOS6 Approximate Gate Equivalents: 4,686*
- *CMOS6 Slowest Timing Path: 101.72 (input to register)*

7.2.2 Optimal El Gamal Hardware Verification

The algorithm was functionally verified at both the language and synthesized level. The following diagrams (Figure 7.2) represent the operation of the algorithm as modeled in the application. Both waveform figures represent the inputs and outputs in hexadecimal format. The second waveform presents the computed resulting output of the algorithm.

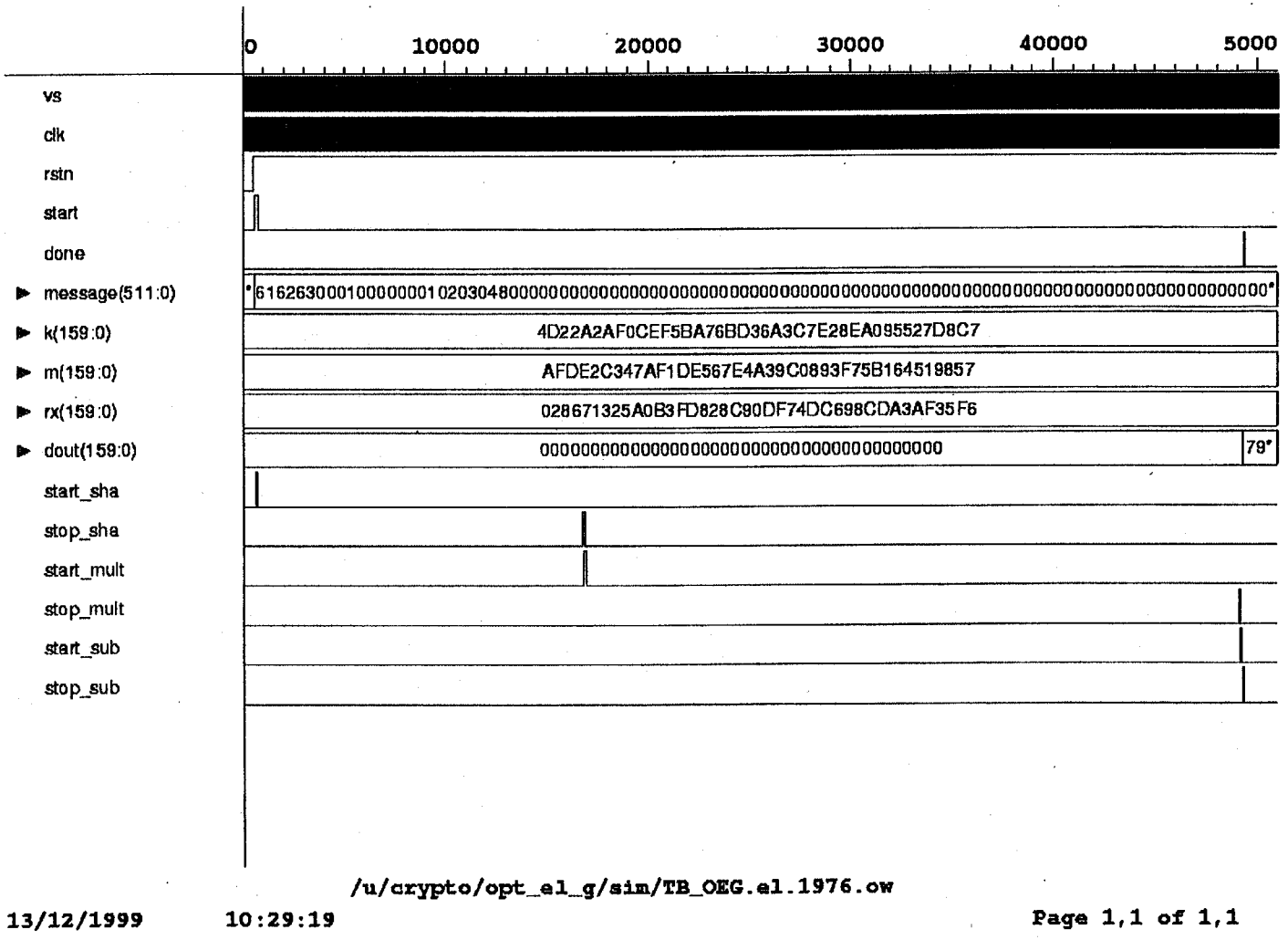

Figure 7.2 Optimal El Gamal Hardware Verification, Inputs and Outputs

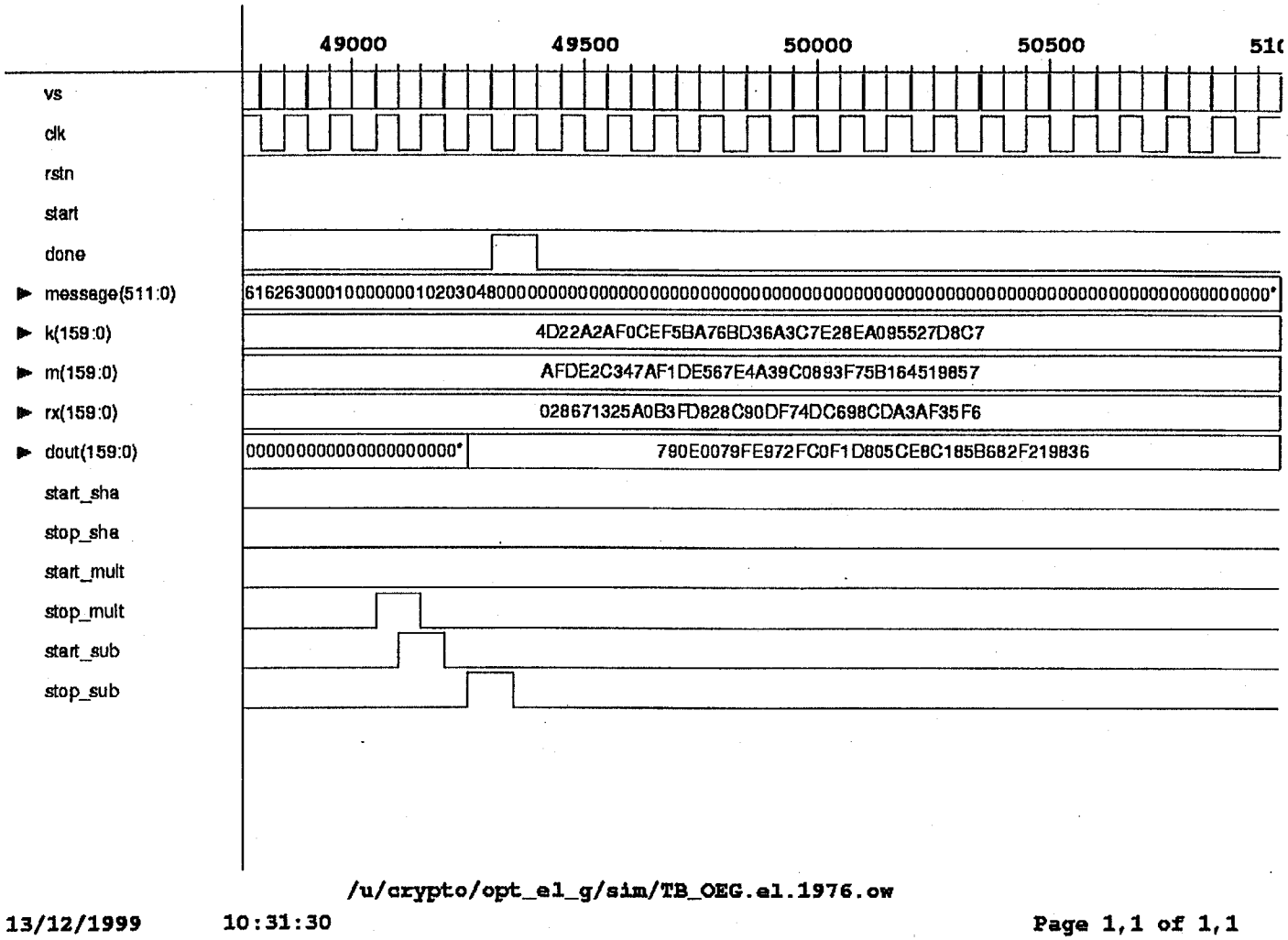

Figure 7.2 Optimal El Gamal Hardware Verification, Inputs and Outputs

7.3 Design of Elliptic Curve Operations and Algorithms

We implemented elliptic curve operations for use with digital signature algorithms. Additionally, we implemented a preliminary version on an Elliptic Curve Optimal El Gamal Digital Signature Algorithm as described below. Refer to Section 2.1.4 for more details.

To generate a signature on a message M:

1. Generate a key pair (u, $V = uG$), where u is a random integer mod r. Let $V = (x_V, y_V)$ ($V \neq O$ because V is a public key)
2. Convert x_V into an integer i
3. Compute an integer $c = i \bmod r$. If $c = 0$, then go to Step 1
4. Set f = Hash (M). Compute an integer $d = (cfs + u) \bmod r$. If $d = 0$, then go to Step 1

Output the pair (c, d) as the signature

The approach for designing the elliptic curve algorithm is similar to the Optimal El Gamal approach. We developed a set of basic mathematical functions, and then used the functions to build the algorithms.

7.3.1 Hardware Implementation of Basic Mathematical Elliptic Curve Functions

We developed a set of basic mathematical functions in VHDL, which was used to build the elliptic curve algorithms. The operations were developed for both 89-bit and 178-bit numbers for elliptic curves defined over $GF(2^{89})$ and $GF(2^{178})$, respectively. First, the basic functions were implemented over the underlying finite field. The elliptic curve operations were then defined in terms of these finite field operations. Finally, the signature algorithm was developed.

7.3.1.1 Addition/Subtraction over $GF(2^m)$

Addition and subtraction on $GF(2^m)$ are the same operation, a bitwise XOR of the 2^m bit vectors. That is, $a + b$ or $a - b$ is equivalent to the bitwise logical function a XOR b. The operation is defined as follows:

$$a, b \in F \quad a + b = (a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_m \oplus b_m)$$

7.3.1.2 Multiplication over $GF(2^m)$

Multiplication in $GF(2^m)$ is essentially the same Shift and Add Algorithm used for binary numbers, followed by a mod (P) operation, where P is the irreducible degree m polynomial defining the field. Essentially, it is like multiplication without any carries, where the accumulation is performed as described above (i.e., XOR function).

7.3.1.3 Division over *GF*(2^m)

Division over *GF* (2^m) is performed by an inversion followed by a multiply. The algorithm we implemented for this function is the Almost Inverse Algorithm developed by co-author Rich Schroppel. A summary of the algorithm follows:

```
A is the input value to be inverted.

P is the irreducible polynomial for the Galois Field.

INT K=0;

GF B=1, C=0, F=A, G=P;

Tag:

while (even(F) {
          F = F / u;      // shift right
          C = C * u;      // Left shift
          K++;
     }
     if(F=1)
          GOTO Done;
     if(Degree(F)<Degree(G)){
          tmp = F;  // Swap F & G
          F = G;
          G = tmp;
          tmp = B;  // Swap B & C
          B = C;
          c = tmp;
     }
     F = F + G;
     B = B + C;
     GOTO Tag;
```

```
Done:
     for(i=0; i<K; i++) {
          //divide B by U^K
          B = B + B(0)*P; // this clears out the LSB
                    // by adding it modulo P
          B = B / u ;     // right shift
     }
     Return B;
```

7.3.1.4 Squaring over $GF(2^m)$

The squaring operation over $GF(2^m)$ is performed by treating the field as a bit vector and inserting a zero in every other place making it twice as big. The new value is then reduced modulo the field polynomial P. For example, the square of the field 1011 is 1000101, which is then reduced modulo the field polynomial P to get the final value.

7.3.1.5 Square Root over $GF(2^m)$

The square root function is somewhat more complex. The even bits of the field are split off to form a new vector, as are the odd bits. The odd bits are then multiplied by a correction factor and added to the vector formed from the even bits. The correction factor is dependent on P.

7.3.1.6 Quadratic Solution (QSolve) over $GF(2^m)$

The function Qsolve finds a solution, z, to the following equation: $z^2 + z = a$.

The QSolve function is linear:

$$QS(A+B) = QS(A) + QS(B)$$

Given that Qsolve is linear, we can derive the $QS(2^m)$ for each bit in the field, and then add them together as appropriate.

7.3.1.7 Arithmetic over $GF(2^{89})$ and $GF(2^{178})$

First, the basic arithmetic operations described above were implemented and tested over the field $GF(2^{89})$. From there, it was easy to adapt the algorithms to $GF(2^{178})$. Since 178 = 89*2, we used field towers (see Section 2.7.2) to think of $GF(2^{178}) = GF(2^{89})^2$ as a degree two field extension over $GF(2^{89})$. An element $\alpha \in GF(2^{178})$ can be represented as a pair of elements in

$GF(2^{89})$: $\alpha = (a,b) : a,b \in GF(2^{89})$. The mathematical operations on $GF(2^{178})$ can thus be defined in terms of the $GF(2^{89})$ routines.

7.3.1.8 Multiplication over $GF(2^{178}) = GF((2^{89})^2)$

The Multiplication Algorithm is defined in terms of the base operations over the $GF(2^{89})$. For example, in the equation below, $a + b$ is over $GF(2^{89})$. The $GF(2^{178})$ Multiplication Algorithm was implemented as follows:

Input: 2 elements $\omega, \sigma \in GF(2^{178})$ of the form $\omega = (a,b); \sigma = (c,d)$

where

$a,b,c,d \in GF(2^{89})$.

Output: $\omega * \sigma = \mu = (e,f); e,f \in GF(2^{89})$

1. $e = (a+b)(c+d) - bd$
2. $f = ac + bd$
3. Output $\mu = (e,f)$

7.3.1.9 Inversion in $GF(2^{178}) = GF((2^{89})^2)$

Similarly, the Inversion Algorithm implemented in VHDL is as follows. Again, base operations are over $GF(2^{89})$.

Input: 1 element $\omega \in GF(2^{178})$ of the form $\omega = (a,b)$ where $a,b \in GF(2^{89})$

Output: $\omega^{-1} = (c,d); c,d \in GF(2^{89})$

1. $c = \dfrac{a}{(a+b)^2 + ab}$
2. $d = \dfrac{a+b}{(a+b)^2 + ab}$
3. Output $\omega^{-1} = (c,d)$

7.3.1.10 Squaring in $GF(2^{178}) = GF((2^{89})^2)$

The Squaring Algorithm for $GF(2^{178})$ was implemented in VHDL as follows. The squaring of $c=a^2$ below is simply an implementation of the squaring for $GF(2^{89})$.

Input: $\omega = (a,b) \in GF(2^{178}); a,b \in GF(2^{89})$

Output $\omega^2 = (c,d)$

1. $c = a^2$
2. $d = a^2 + b^2$
3. Output $\omega^2 = (c,d)$

7.3.1.11 Square Root in $GF(2^{178}) = GF((2^{89})^2)$

The Square Root Algorithm for $GF(2^{178})$ was implemented in VHDL as follows. As in the squaring, the square root of a is the exact implementation in $GF(2^{89})$.

Input: $\omega = (a,b) \in GF(2^{178}); a,b \in GF(2^{89})$

Output: $\sqrt{\omega} = (c,d)$

1. $c = \sqrt{a}$
2. $d = \sqrt{a} + \sqrt{b}$

7.3.1.12 Quadratic Solve (Qsolve) in $GF(2^{178}) = GF((2^{89})^2)$

The Square Root Algorithm for $GF(2^{178})$ was implemented in VHDL as follows:

Input: $\omega = (a,b) \in GF(2^{178}); a,b \in GF(2^{89})$

Output: $QSolve(\omega) = \alpha = (c,d)$ (Here $\alpha^2 + \alpha = \omega$)

1. $c = QSolve(a)$
2. $d = QSolve(a + b + c)$
3. Set $z = a + b + c = z_0 z_1 ... z_{88}$
4. If $z_0 \oplus z_{51} = 1$ then $c = c + 1$
5. Output (c,d)

7.3.2 Elliptic Curve Operations

With the base finite field operations in place, we are ready to implement the elliptic curve operations.

7.3.3 Elliptic Curve Scalar Multiplication Algorithm

The elliptic curve scalar multiplication algorithm consists of two basic operations: point addition and point halving, as shown in the following block diagram (Figure 7.3). The Point Addition and Point Halving algorithms use all of the basic finite field mathematical operations just described.

The Elliptic Curve Scalar Multiplication Algorithm used for our VHDL implementation is as follows:

Input: $P=(x,y)$, an elliptic curve point on E, and $n = k_{178}...k_0$, the binary representation of a random number with ≤ 178-bits

Output: nP, the scalar multiplication

$V=0$

For $I = 0$ to 178

do{

If $k_i = 1$ then

$V=V+P$ —Point Addition Algorithm

End if;

V= ½ V

} —Point Halving Algorithm

end loop;

Note that the above algorithm really produces $nP/2^{178}$. The exact factor is recoverable later, if needed.

The block diagram (Figure 7.3) is as follows:

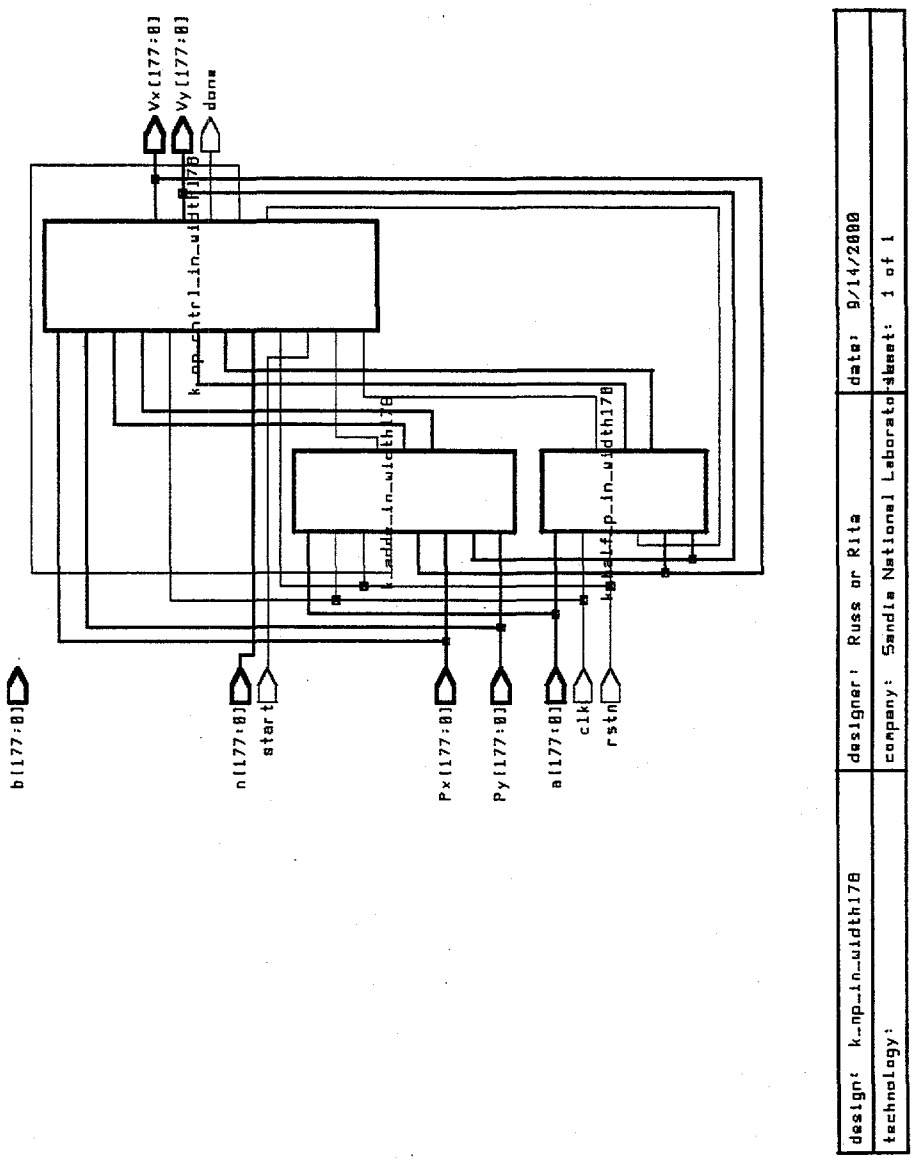

Figure 7.3 Elliptic Curve Scalar Multiplication

7.3.3.1 *NP* Control Block

The *nP* control block provides the control for the algorithm. It is essentially a state machine implementation that directs the operation sequence.

7.3.3.2 Point Addition over $GF(2^m)$

The algorithm for addition on E [P1363] that was implemented in VHDL follows:

Input: $a, b, P_0 = (x_0, y_0), P_1 = (x_1, y_1)$

Output: $P_2 = (x_2, y_2)$

1. If $P0 = O$, (i.e. $x_0 = 0$ and $y_0 = 0$) then output $P2 \leftarrow P1$ and stop

2. If $P_1 = O$, then output $P_2 \leftarrow P_0$ and stop

3. If $x_0 \neq x_1$ then

   Set $\lambda \leftarrow (y_0 + y_1) / (x_0 + x_1)$

   Set $x_2 \leftarrow a + \lambda^2 + \lambda + x_0 + x_1$

   Go to step 7

4. If $y_0 \neq y_1$ then output $P_2 \leftarrow O$ and stop

5. If $x_1 = 0$ then output $P_2 \leftarrow O$ and stop

6. Set

   $x_1 + y_1 / x_1$

   $x_2 \leftarrow a + \lambda^2 + \lambda$

7. $y_2 \leftarrow (x_1 + x_2)\,\lambda + x_2 + y_1$

8. $P_2 \leftarrow (x_2, y_2)$

Essentially, this algorithm required 2 elliptic curve multiplications, a squaring, and an inversion. The algorithm was verified via VHDL testbenches.

7.3.3.3 Point Halving over $GF(2^m)$

The Point Halving Algorithm (patent pending) developed by Rich Schroeppel [S00], is, in software, three times as fast as point doubling. It therefore provides a dramatic increase in performance when used in computing a multiple of a point.

We implemented the following algorithm for point halving:

1. Convert point from (Px,Py) to (Px,Pr)

   $Ptmp$ = Invert Px

   $Pr = Ptmp * Py$

2. Mh = QSolve($Px + A$), where A is a curve parameter

3. $T = Px*(Pr + Mh)$

4. If parity(T and tm)=0 then $Mh = Mh + 1$; $T = T + Px$; end if;

tm is a mask that is dependent upon the modulus polynomial.

In our case, it is u^51 + 1.

5. Pxh = square_root(T)
6. $Prh = Mh + Pxh + 1$
7. Convert point from (Px,Pr) to (Px,Py)
8. $Pyh = Prh * Pxh$

This algorithm can be optimized in its present state. In the NP Algorithm, one does a point addition only for each set bit in the integer multiple, but does a point halve for each bit. Therefore, if (Px,Pr) is carried and converted in point add to (Px,Py), one gets a net savings.

7.3.3.4 Scalar Multiplication Verification

The Scalar Multiplication Algorithm for both $GF(2^{89})$ and $GF(2^{178})$ were functionally verified at the VHDL language level. The following diagram (Figure 7.4) represents the operation of the $GF(2^{178})$ Algorithm as modeled in the application. The input and output signals are represented in hexadecimal format.

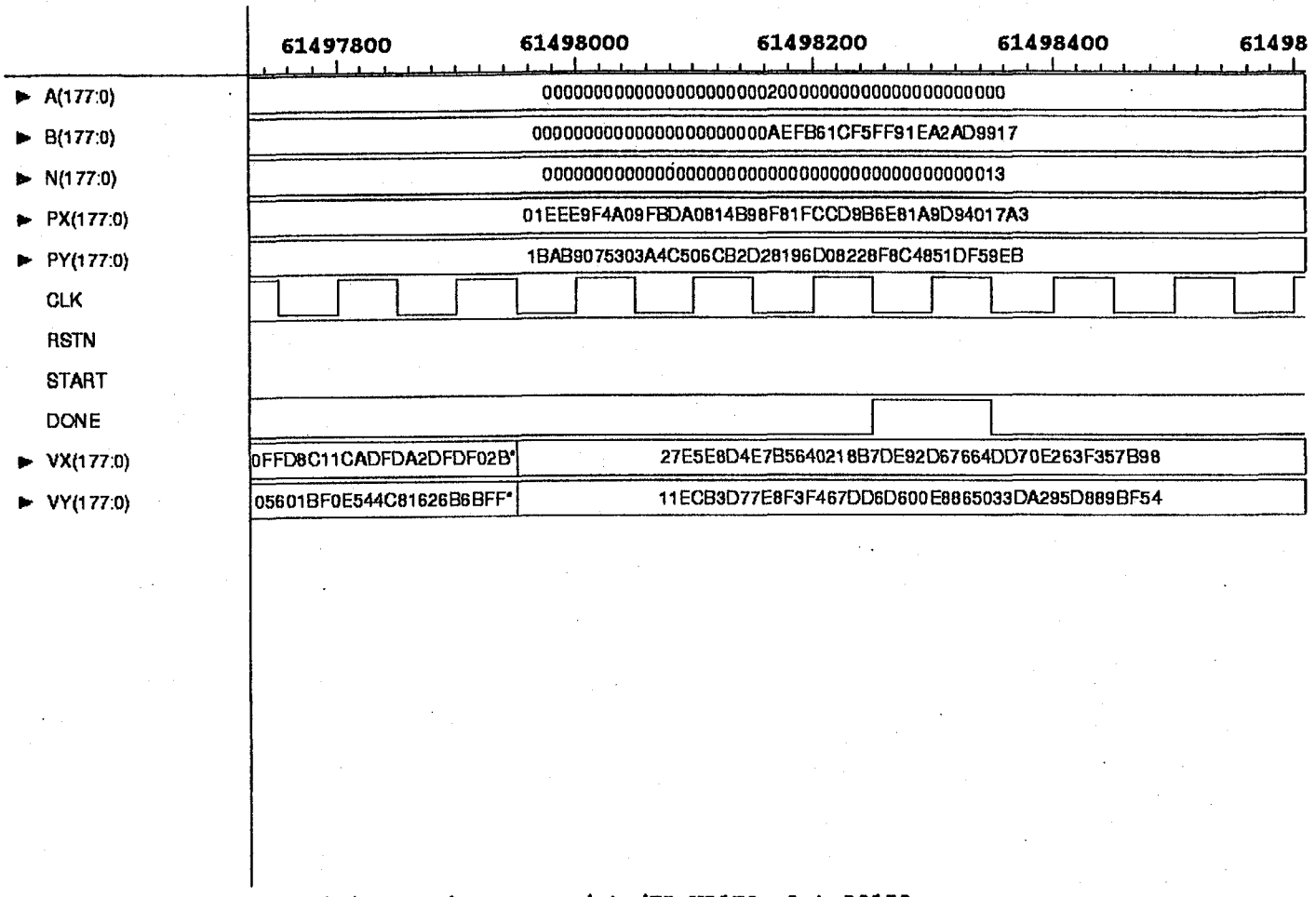

Figure 7.4 Multiplication in $GF(2^{178})$

7.4 Obtaining the VHDL Code for Implemented Functions

One advantage of modeling and capturing the design in VHDL is that it is easy to target or re-target any ASIC, FPGA, or PLA library with very little effort. Thus, all of the VHDL code described in the previous sections is available from the authors of this document.

7.5 Comments and Further Optimizations

In the current hardware implementation of the Halving Algorithm (Section 7.3.3.3), the ratio R is computed for each point halving, and then the resulting Rhalf is converted back to Yhalf. This is reasonable for proof of concept, but it takes a reciprocal and a multiply to compute R, and a multiply to compute Y. If these are removed, the halving steps should be four times as fast. Furthermore, in the transition from a halving step to a point addition, the Y coordinate is computed for the addition. This is unavoidable. But the output of the point addition can be in (X,R) form, which saves some work in preparing for the next halving step; or, the point halving can start from XY format, and output in XR.

The Point Halving Algorithm is more than three times as fast as point doubling in software, and we expect comparable performance in optimized hardware. Even after the other parts of the overall point-multiplication process are accounted for, the performance gain from using point halving is roughly 2.5. In an optimized implementation, a table of small odd multiples of the generator is computed at the start. This reduces the number of point additions considerably, so that five doubling (or halving) steps are done for every addition. This allows the full benefit of the halving improvement to be realized.

A point addition requires three expensive operations in the underlying field: two multiplications and a reciprocal. The reciprocal is typically three times as expensive as the multiplication, so a simple cost estimate for point addition is five times the cost of a field multiplication. Point doubling is similar. Point halving requires only a single multiplication, plus some auxiliary operations—a square root and solving a quadratic equation. The auxiliary operations are relatively cheap in $GF(2)$ fields, so the overall cost of point halving is about 1.5 multiplications.

An additional benefit of using fewer reciprocal operations is that multiplication is easier to improve: the Karatsuba method can be used to speed up multiplication. Nothing as effective is known for reciprocals.

We anticipate further improvements in the circuits for square root and for solving the quadratic equation.

We should also mention the benefit of our field-tower implementation. The multiplication using field towers is nearly 4/3 as fast as a straightforward multiplication in a double-length field. In effect, we get the benefit of one level of Karatsuba optimization. The reciprocal operation is considerable faster than a straightforward double-length reciprocal, since only half as many clock steps are used, with only single-length operands.

8. References

[A88] G.B. Andrew, 1987, Random Sources for Cryptographic Systems, in *Advances in Cryptology—EUROCRYPT '87 Proceedings*. Springer-Verlag.

[BCCG92] T. Baritaud, M. Campana, P. Chauvaud, and H. Gilbert, 1992, On the Security of the Permuted Kernel Identification Scheme, in *Proceedings of CRYPTO '92*. Springer-Verlag.

[BGG94] M. Bellare, O. Goldreich, and S. Goldwasser, 1994, The Case of Hashing and Signing, in *Proceedings of CRYPTO '94*. Springer-Verlag.

[BPV98] V. Boyko, M. Peinado, and R. Venkatesan, 1998, Speeding up Discrete Log and Factoring Based Schemes via Precomputations, in *Proceedings of EUROCRYPT '98*. Springer-Verlag.

[CB95] A. Chandrakasan, and R. Brodersen, 1995, Minimizing Power Consumption in Digital CMOS Circuits, in *Proceedings of the IEEE*, Vol. 83, No. 4, April.

[DK90] S. R. Dusse and B. S. Kaliski Jr., 1990, A Cryptographic Library for the Motorola DSP56000, in *Proceedings of EUROCRYPT '90*. Springer-Verlag.

[dR93] P. deRooij, 1993, On Schnorr's Pre-processing for Digital Signature Schemes, *Proceedings of EUROCRYPT '93*. Springer-Verlag.

[FMC84] R.C. Fairfield, R.L. Mortenson, and K.B. Coulthart, 1984, An LSI Random Number Generator, in *Advances in Cryptology—CRYPTO '84 Proceedings*. Springer-Verlag.

[G98] D. M. Gordon, 1998, A Survey of Fast Exponentiation Methods, in *Journal of Algorithms*, No. 27, pg. 129-146.

[HX94] L. Harn and Y. Xu, 1994, Design of Generalised El Gamal Type Digital Signature Schemes Based on Discrete Logarithm, in *Electronics Letters Online*, No.19941398, September 30. IEEE.

[K81] D. E. Knuth, 1981, The Art of Computer Programming, in *Seminumerical Algorithms*, 2nd ed., vol 2. Addison-Wesley, Reading, MA.

[L97] M. Levy, Ed., 1997, μprocessor/μcontroller Directory. EDN, September 25.

[MI88] T. Matsumoto and H. Imai. Public Quadratic Polynomial-tuples for efficient signature-verification and message-encryption, *Proceedings of EUROCRYPT'88*, Springer-Verlag.

[MvV97] A. Menezes, P. van Oorschot, and S. Vanstone, 1997, Handbook of Applied Cryptography. CRC Press.

[MS94] W. Meier and O. Staffelbach, 1994, The Self-Shrinking Generator, in *Advances in Cryptology—EUROCRYPT '94 Proceedings*. Springer-Verlag.

[RML97] A. Royo, J. Moran, and J. Lopez, 1997, *Design and Implementation of a Coprocessor for Cryptography Applications*, IEEE .

[NR94] K. Nyberg and R. A. Ruppel, 1994, Message Recovery for Signature Schemes Based on the Discrete Logarithm Problem, in *Pre-proceedings of EUROCRYPT '94*, May.

[P95] D. Pointcheval, 1995, A New Identification Scheme Based on the Perceptrons Problem, in *Proceedings of EUROCRYPT '95*. Springer-Verlag.

[P1363] IEEE P1363, Standard Specifications for Public Key Cryptography. Appendix A.

[Pa95] J. Patarin, 1988, Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt '88, in *Proceedings of CRYPTO '95*. Springer-Verlag.

[Pa96] J. Patarin, 1996, Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new Families of Asymmetric Algorithms, in *Proceedings of EUROCRYPT'96*. Springer-Verlag.

[PC93] J. Patarin and P Chauvaud, 1993, Improved Algorithms for the Permuted Kernel Problem, in *Proceedings of CRYPTO '93*. Springer-Verlag.

[S89] C. Schnorr, 1989, Efficient Identification and Signatures for Smart Cards, in *Proceedings of CRYPTO '89*. Springer-Verlag.

[S00] R. Schroeppel, 2000, Elliptic Curves–Twice as Fast, Rump Session Talk at *CRYPTO '00*. Springer-Verlag.

[S89a] A. Shamir, 1989, An Efficient Identification Scheme Based on Permuted Kernels, in *Proceedings of CRYPTO '89*. Springer-Verlag.

[S93] J. Stern, 1993, A New Identification Scheme Based on Syndrome Decoding, in *Proceedings of CRYPTO '93*. Springer-Verlag.

[S94] J. Stern, 1994, Designing Identification Schemes with Keys of Short Size, in *Proceedings of CRYPTO '94*. Springer-Verlag.

[Sil86] J. Silverman, 1986, *The Arithmetic of Elliptic Curves*. Springer-Verlag.

[T99] Tundra Semiconductor Corporation, *RGB1210 Data Sheet*. Tundra Semiconductor Corporation.

9. Appendix A—Additional Algorithms

9.1 Storage/Work Balanced El Gamal Scheme

This scheme is a modification of the Optimal El Gamal scheme. It requires less storage and only two more modular multiplications to sign. It introduces a secret parameter d that allows values of k_i to be recovered simply. Each $k_i = dk_{i-1} \bmod q$ is computed as needed in the generation of s_i. All values of $r_i = (g^{k_i} \bmod p) \bmod q$ are pre-computed and stored. Therefore, at any given time, the values that are stored include d, q, p, g, k_{i-1}, and all values of r. The values of r need not be kept secret. The signature for message i is r_i, s_i as computed in the Optimal El Gamal scheme. When the values of r are exhausted, new values must be computed, requiring that a modular exponentiation or another authentication mechanism be used.

This implementation of the El Gamal signature scheme would have to store only one 20-byte pre-computed value for each message it is to sign, and when this value has been used, the application could no longer be able to sign messages. It would then have to use some other mechanism to sign messages.

Example: (20 bytes) * (1 message/day) * (365 days/year) * (5 years) = 36,500 bytes of pre-computed data

9.1.1 Evaluation of Storage/Work El Gamal Scheme

Table 9.1 Parameter Sizes for Storage/Work El Gamal Scheme

Parameter	Description	Size (in bits)
P	Prime modulus	≥ 768
Q	Prime divisor of p	≥ 160
G	Any value such that order of g is q	≥ 768
X	Secret key	160
Y	$g^x \bmod p$	≥ 768
H(m)	Hash digest of the message	160
D	Additional secret	160
k_I	Random per message secret, $k_i = dk_{i-1} \bmod q$	160
r_I	$(g^{k_i} \bmod p) \bmod q$	160
s_I	$(r_i x H(m) - k_i) \bmod q$	160

Signature operations required: Hash of message, two 160-bit modular multiplications and subtraction

Verification operations Hash of message, 160-bit modular multiplication,

required:	two modular exponentiations in the size of p
Storage required to sign (in bits):	$768 \leq p$, g, 160 bits each for q, d, k_{i-1} and x, and 160 bits for each message to be signed
Amount of data transmitted:	40 bytes

9.2 Schnorr Scheme

The Schnorr scheme [S89] is a DSA variant that takes advantage of a pre-processing mechanism. However, the pre-processing mechanism presented by Schnorr has been successfully attacked [dR93]. The algorithm is described below.

To generate a key pair:

1. Assume system parameters identical to those defined for the Digital Signature Algorithm
2. Choose a random number $x \bmod q$ where x is the private key, and compute the public key $y=g^{-s} \bmod p$

To generate a signature:

1. Using the pre-processing procedure described below, pick a random number $k \in \{1,\ldots,q\}$, and compute $r' = g^k \bmod p$
2. Compute $r = Hash(r', m)$
3. Compute $(s = k + xr) \bmod q$, and output the signature (r,s)

To verify a signature:

Compute $\bar{r} = g^s y^r$, and verify that $r = Hash(\bar{r}, m)$

Pre-processing procedure:

1. Generate independent random pairs $(k_i, r_i'), i = 1,\ldots j \exists r_i' = g^{k_i} \bmod p$
2. For every signature use a random combination (k,r') of these pairs
3. Rejuvenate the collection of these pairs by combining randomly selected pairs
4. The algorithm to chose random combinations must remain secret

As stated above, this pre-processing algorithm as specifically described by Schnorr has been proved to be insecure. However, the proof does not show that the idea of pre-processing in and of itself is insecure, only that the method described by Schnorr is insecure. Unfortunately, the

requirement that the pre-processing algorithm remain secret may be an undue restriction for certain applications.

9.3 Chor-Rivest Knapsack Signature Scheme

The following description of the Chor-Rivest public key encryption scheme is from [MvV97].

Chor-Rivest Key Generation:

1. Select a finite field F_q of characteristic p, where $q = p^h, p \geq h$, and for which the discrete logarithm problem is feasible
2. Select a random monic irreducible polynomial $f(x)$ of degree h over Z/pZ. The elements of F_q are represented as polynomials in Z/pZ [x] of degree less than h, with multiplication performed modulo $f(x)$
3. Select a random primitive element $g(x)$ of the field F_q
4. For each ground field element $i \in \mathsf{Z}/\mathsf{pZ}$, find the discrete logarithm $a_i = \log_{g(x)}(x+i)$ of the field element $(x+i)$ to the base $g(x)$
5. Select a random permutation π on the set of integers $\{0,1,2,\ldots,p-1\}$
6. Select a random integer $d, 0 \leq d \leq p^h - 2$
7. Compute $c_i = \left(a_{\pi(i)} + d\right) \bmod (p^h - 1), 0 \leq i \leq p-1$
8. The public key is $((c_0, c_1, \ldots, c_{p-1}), p, h)$ and the private key is $(f(x), g(x), \pi, d)$

To encrypt a message:

1. Represent message m as a binary string of length $\left\lfloor \lg \binom{p}{h} \right\rfloor$ where $\binom{p}{h}$ is a binomial coefficient
2. Transform m into a binary vector $M = (M_0, M_1, \ldots, M_{p-1})$ of length p having exactly h ones as follows:

   set $l \leftarrow h$

   for i from 1 to p

if $m \geq \binom{p-i}{l}$ then set $M_{i-1} \leftarrow 1, m \leftarrow m - \binom{p-i}{l}, l \leftarrow l-1$

else $M_{i-1} \leftarrow 0$

3. Compute $c = \sum_{i=0}^{p-1} M_i c_i \bmod \left(p^h - 1\right)$

To decrypt a message:

1. Compute $r = (c - hd) \bmod \left(p^h - 1\right)$
2. Compute $u(x) = g(x)^r \bmod f(x)$
3. Compute $s(x) = u(x) + f(x)$, a monic polynomial of degree h over Z / pZ
4. Factor $s(x)$ into linear factors over $\mathsf{Z}_p \left| s(x) = \prod_{j=1}^{h} (x + t_j) \right.$, where $t_j \in \mathsf{Z/pZ}$
5. Components of the vector M that are one have indices $\pi^{-1}(t_j), 1 \leq j \leq h$. All other components are zero
6. The message m is recovered by

set $m \leftarrow 0, l \leftarrow h$

for i from 1 to p

if $M_{i-1} = 1$ then set $m \leftarrow m + \binom{p-i}{l}$ and $l \leftarrow l-1$

This description of the Chor-Rivest scheme defines p as prime. However, Z / pZ can be replaced by a field of prime power order. In addition, in order to feasibly compute discrete logarithms, the parameters p and h must be chosen so that $q = p^h - 1$ has only small factors. The recommended size of the parameters are $p \approx 200$ and $h \approx 25$. Unfortunately, this causes the public key to be roughly 40,000 bits in length, making this algorithm not feasible for the low-power environment.

9.4 McEliece Scheme

The McEliece Public Key Encryption Algorithm [MvV97] is based on the difficulty of decoding an arbitrary linear code which is known to be NP-hard. It has received little practical attention due to the size requirements of public keys. The following text describes the encryption algorithm, not the authentication algorithm.

To generate a key pair:

1. Choose system parameters k, n, and t (recommended values $n = 1024$, $t = 38$, and $k \geq 644$ when Goppa codes are used as the error correcting code for which efficient decoding algorithms are known)

2. Choose a $k \times n$ generator matrix G for a binary (n, k) linear code which can correct t errors and for which an efficient decoding algorithm is known

3. Select a random $k \times k$ binary non-singular matrix S

4. Select a random $n \times n$ permutation matrix P

5. Compute the $k \times n$ matrix $\hat{G} = SGP$

6. An entity's public key is $(\hat{G}, t)$. The corresponding private key is (S, G, P)

To encrypt a message:

1. Represent the message as a binary string m of length k

2. Choose a random binary error vector z of length n having at most t ones

3. Compute ciphertext, the binary vector $c = m\hat{G} + z$

To decrypt a message:

1. Compute $\hat{c} = cP^{-1}$

2. Use the decoding algorithm for the code generated by G to decode $\hat{c}$ to $\hat{m}$

3. Compute $m = \hat{m}S^{-1}$

The size of the public key for $n = 1024$ and $k \geq 644$ would be 82 kilobytes. The size of the private key is 264 kilobytes.

10. Appendix B

The purpose of this appendix is to present some ideas for converting the Permuted Kernel and the Syndrome Decoding identification schemes to signature schemes suitable to the low-powered environment. The ideas are incomplete and are presented solely for informational purposes. Our hope is that the ideas may be useful to the reader in understanding the problems that he may improve upon our solutions.

10.1 Permuted Kernel Problem Scheme

This public key identification scheme is defined in the abstract [S89a] with further analysis done in [BCCG92] and [PC93]. This scheme is based on an NP-complete algebraic problem known at the permuted kernel problem. The problem is defined as [MvV97]:

Given: an $m \times n$ matrix A over Z / pZ, p prime and relatively small (e.g., 251)

an n-vector V

Find: a permutation π on $\{1, ..., n\}$ such that $V_\pi \in Ker(A)$

Where: $Ker(A)$ is defined as the kernel of A consisting of all n-vectors W such that $AW = [0 ... 0] \bmod p$

To generate a key pair:

- all users agree on a matrix A and a prime p
- each user chooses a random permutation π_i as his private key and a random vector $V \ni V_{\pi_i} \in Ker(A)$ which serves as his public key

Permuted Kernel Problem Progress Towards Possible Solution

Given: a prime number p

an m x n matrix $A=(a_{ij}) \in Z_p, i=1...m, j=1...n$

an n-vector $V=(V_j) \in Z_p, j=1...n$

Find: a permutation $\pi(1...n)$ such that $A \times V_\pi = 0, V_\pi = (V_{\pi(j)}), j=1...n$

Key Generation:

- users of the system agree to a system prime p and a matrix $A=(a_{ij}) \in Z_p, i=1...m, j=1...n$
- each user then chooses a permutation $\pi(1...n)$ as his private key and a random n-vector $V=(V_j) \in Z_p, j=1...n$ as his public key such that $A \times V_\pi = 0, V_\pi = (V_{\pi(j)}), j=1...n$, i.e. V_π is in the kernel of A

Use in a Three-Pass Zero Knowledge Identification Scheme (Shamir):

1. The prover (**A**) chooses a random n-vector R and a random permutation σ, and sends the cryptographically hashed values of the pairs (σ, AR) and $(\pi\sigma, R_\sigma)$ to the verifier (**B**).
2. **B** chooses a random value $0 \le c < p$ and asks that **A** send $W = R_\sigma + c(V_\pi)_\sigma$.
3. After receiving W, **B** asks **A** to reveal either σ or $\pi\sigma$. In the first case, **B** checks that $(\sigma, A_\sigma W)$ hashes to the first given value and in the second case, **B** checks that $(\pi\sigma, W - c(V_\pi)_\sigma)$ hashes to the second given value.

Note:

$$A_\sigma W = A_\sigma (R_\sigma + c(V_\pi)_\sigma) = A(R + cV_\pi) = AR$$
$$W - c(V_\pi)_\sigma = R_\sigma$$

The probability that a cheater can evade detection is ½ so the protocol is repeated k times to reduce the probability of successful cheating to some acceptable limit, i.e., $1/2^k$.

Use in a Signature Scheme:

The following is an attempt to modify the identification scheme to a signature scheme suitable for a low-powered environment.

Additional Given:

1. An algorithm, f_1(message, private key), that will produce k random vectors R_i, i=1,2, …, k, each of size n where the elements of R_i are in Z_p. This algorithm is public, but the key is private to **A** (necessary for non-repudiation).
2. An algorithm, f_2(message), that will produce k random values c_i, i=1,2, …, k, where $0 \le c_i < p$. This algorithm is public.
3. An algorithm, f_3(message), that will produce k random permutation vectors, τ_i, i=1,2, …, k. Each vector is a permutation of the integers from 1 to n. This algorithm is public.

4. An algorithm, f_4(message), that will produce k random bits, b_i, i=1,2, …, k. This algorithm is public.

To sign a message:

1. The prover (**A**) computes *wxor* and *hxor* as follows: Given $(R_i, c_i, \tau_i, b_i)\ \forall_i$

Initialize *wxor* and *hxor*

For i=1:k

Case 1: $b_i = 0$

$\sigma_i = \tau_i$

$h = H(\sigma_i, AR_i)$

Case 2: $b_i = 1$

Compute $\sigma_i = g_i(\tau_i)\ \ni (\pi)_{\sigma_i} = \tau_i$ KEY (uses knowledge of π to get σ_i)

$h = H(\tau_i, (R_i)_{\sigma_i})$

$wxor = wxor \oplus \left((R_i)_{\sigma_i} + c_i (V_\pi)_{\sigma_i}\right)$

$hxor = hxor \oplus h$

end;

2. The prover (**A**) sends (message, *hxor*[commitment], *wxor*[signature]).

To verify a message:

1. The verifier (**B**)

computes $(c_i, \tau_i, b_i)\ \forall_i$

initialize *hxorv*

For i=1:k

Case 1: $b_i = 0$

[Note in this case $\tau_i = \sigma_i$ w.r.t prover]

$h = H(\tau_i, A_{\tau_i} wxor)$

Case 2: $b_i = 1$

$h = H(\tau_i, wxor - c_i V_{\tau_i})$

$hxorv = hxorv \oplus h$

end;

2. Verify *hxorv=hxor*

Comment: We note that this idea does not work since the *wxor* will never check properly. We suggest the following modifications.

MODIFICATION 1

Case 1: $b_i = 0$ σ_i known by verifier

Compare: $AR_i = A_{\sigma_i} W_i$

- FOR LHS, we send selected bits of $\sum_i AR_i \bmod(p)$, where the selected bits are determined by the message.
- FOR RHS, observe that $A_{\sigma_i} W_i = A_{\sigma_i}\left[(R_i)_{\sigma_i} + c_i (V_\pi)_{\sigma_i}\right]$, or $A_{\sigma_i} W_i = A_{\sigma_i}(R_i)_{\sigma_i} + A_{\sigma_i} c_i (V_\pi)_{\sigma_i}$, or $A_{\sigma_i} W_i = AR_i + Ac_i V_\pi = AR_i$.

  Seems like W_i would be an important key (maybe send message dependant bits of W) ---still BIG PROBLEM

A composite of W_i is not sufficient because of interaction between W_i and A_i

Case 2: $b_i = 1$ π_{σ_i} known by verifier

Compare: $(R_i)_{\sigma_i} = W_i - c_i(V_\pi)_{\sigma_i}$

- FOR LHS, we send selected bits of $\sum_i (R_i)_{\sigma_i} \bmod(p)$
- FOR RHS, we send selected bits of $\sum_i W_i \bmod(p)$, note that $c_i(V_\pi)_{\sigma_i}$ can be computed by the verifier

MODIFICATION 2

The problem with the above scheme is that in the verification procedure, B must use as input to H a quantity related to the $w_i = ((R_i)_{\sigma_i} + c_i(V_\pi)_{\sigma_i})$ which cannot be recovered from the wxor. Instead of the wxor we suggest in the signing procedure to construct a composite sum, $W = \sum_i d_i w_i$, where the d_i's are constructed as follows. Let $p_1, p_2, ..., p_k$ be k distinct primes where $p_i > p$ for all i. These primes can be public, say the next k primes after p. Define

$$n_i = \prod_{j \neq i} p_i \text{ , and let } d_i = n_i(n_i^{-1} \bmod p_i).$$

Note that

$$d_i \equiv 0 \bmod p_j \quad (\text{for all } j \neq i), \text{ and}$$

$$d_i \equiv 1 \bmod p_i,$$

hence,

$$W \bmod p_i \equiv \sum_i d_i w_i \bmod p_i = w_i \bmod p_i$$

and since $p_i > p$ we recover the original w_i. Now in the verification procedure B uses $(W \bmod p_i)$ instead of wxor in the ith loop and the algorithm will then give the desired result. This solution works, however, W is still too large for the low-powered environment. An idea to improve this is given next.

A Partial Solution to the PKP Signature Problem

1. A computes *full* commitments: $\left\{D_1^{(1)}, D_2^{(1)}, D_1^{(2)}, D_2^{(2)}, \ldots, D_1^{(k)}, D_2^{(k)}\right\}$, where

$D_1^{(i)} = H(\sigma_i = \tau_i, AR_i)$; $\tau_i = f_{1i}(message)$, $R_i = f_{2i}(message)[private]$

$D_2^{(i)} = H\left(\pi\sigma_i = \tau_i, (R_i)_{\sigma_i}\right)$; $\tau_i = f_{1i}(message)$, $R_i = f_{2i}(message)$,

$\sigma_i = g_i(\tau_i)$ $\ni (\pi)_{\sigma_i} = \tau_i$ (uses knowledge of π to get σ) (At this point, A has committed to the R_i's)

2. A keeps only the first q bits of each $D_j^{(i)}$ (e.g., $d_j^{(i)} = \left\{D_{j1}^{(i)}, D_{j2}^{(i)}, \ldots D_{jq}^{(i)}\right\}$) as the *effective* commitments. This impacts the security of the system. The probability of successful counterfeiting in the absence of information about π increases from $\left(\frac{1}{2}\right)^k$ to $\left(\frac{1}{2} + \left(\frac{1}{2}\right)^{q+1}\right)^k$.

3. A computes: $G = hash(d_1^{(1)}, d_2^{(1)}, d_1^{(2)}, d_2^{(2)}, \ldots, d_1^{(k)}, d_2^{(k)}; message)$

4. A obtains c_i and b_i (i =1,2, …, k) from G [Note: A has no control over c_i and b_i]

5. A computes W_i (i =1,2, …, k) and composes W (a compression of W_i's) as above.

6. A communicates $(d_1^{(1)}, d_2^{(1)}, d_1^{(2)}, d_2^{(2)}, \ldots, d_1^{(k)}, d_2^{(k)}; W; message)$ to B

7. B obtains c_i and b_i (i =1,2, …, k)
from $G = hash(d_1^{(1)}, d_2^{(1)}, d_1^{(2)}, d_2^{(2)}, \ldots, d_1^{(k)}, d_2^{(k)}; message)$

8. B decomposes W to obtain W_i's

9. B computes:

$b_i = 0: E_1^{(i)} = H(\sigma_i = \tau_i, A_{\tau_i} W_i)$; $\tau_i = f_{1i}(message)$ (or)

$b_i = 1: E_2^{(i)} = H\left(\tau_i, W_i - c_i V_{\tau_i}\right)$; $\tau_i = f_{1i}(message)$

10. B retains only the first q bits of each $E_j^{(i)}$ (*e.g.*, $e_j^{(i)} = \{E_{j1}^{(i)}, E_{j2}^{(i)}, \ldots E_{jq}^{(i)}\}$) to check versus the respective *effective* commitments from A.

Comment: In this implementation, finding a compression/decompression of W (steps 5 and 8) is clearly the problem. The signature size currently dominated by W. The security of such an approach is unknown.

10.2 Syndrome Decoding Based Identification Scheme (Stern)

This public key identification scheme is defined in [S93]. This scheme is based on the syndrome decoding problem for error-correcting codes. The security of this scheme is based on the hardness of decoding a word of given syndrome with respect to some binary linear error-correcting code.

To generate a key pair:

- all users agree on an $k \times n$ matrix H over F_2 (this matrix can be considered a parity check matrix)
- all users agree on a value $p < n$
- each user picks an n-bit value s which is comprise of p ones
- all users agree on a cryptographic hash function
- each user computes his public key $i = H(s)$

Given: parameters n and k such that $n = 2k, k = 256, 512$

H, a random* $(n-k) \times n$ binary matrix (system parameter)

$p \approx 0.11n$, a weight parameter (system parameter)

s, a binary vector of length n containing exactly p ones (private key)

$i = Hs$, an $(n-k)$ binary vector (corresponding public key)

*Stern defines H to be a random binary matrix and also indicates that it is a linear error correcting code.

Questions: Can H be truly random or are there restrictions on the formation of H such as H must contain columns which are non-zero and distinct? If there are restrictions on the generation of H, is the security of the scheme affected?

Identification Procedure:

1. **A** picks an n-bit vector y and a permutation $\sigma=\{1,2,\ldots,n\}$ and sends **B**

   $c_1=F(\sigma,H(y))$
   $c_2=F(y_\sigma)$
   $c_3=F((y\oplus s)_\sigma)$

2. **B** sends a challenge b = 0, 1, or 2

3. Case b = 0: **A** reveals y and σ

   B verifies c_1 and c_2

   Case b = 1: **A** reveals $y\oplus s$ and σ

   B verifies commitments c_1 and c_3

   note: $H(y)=(H(y\oplus s))\oplus i$

   Case b = 2: **A** reveals y_σ and s_σ

   B verifies commitments c_2 and c_3 and verifies $wt(s_\sigma)=p$

Comment:

The probability that a cheater can evade detection is *2/3* so the protocol is repeated k times to reduce the probability of successful cheating to some acceptable limit, i.e., $2/3^k$.

Use in a Signature Scheme

Method 1 – [S93]

1. **A** prepares k sets of commitments: $C_j=\{c_1^j,c_2^j,c_3^j\}, j=1,2,\ldots,k$

2. **A** hashes commitments with message: $G = hash(c_1^1,c_2^1,c_3^1,\ldots,c_1^k,c_2^k,c_3^k;message)$

3. **A** uses the successive digits of G (converted to a base 3 representation) as the sequence of k challenges (b's).

4. **A** responds to challenges and issues transcript of commitments and responses (signature) along with message to **B**

5. **B** computes *G*, reconstitutes series of challenges, and verifies responses.

Comment:

- Method 1 is likely to result in very long signature.

Method 2 – Incomplete (Attempt to minimize signature size by having **B** pre-compute some bit-strings)

1. **A** and **B** generate m n-dimensional random bit-strings (v_j , $j = 1, 2, ..., m$) using f(message), where f is a public hash function. In a sense, these bit-strings are the commitments.

2. Let $v_j = (y_j \oplus s)_\sigma$ for $j = 1, 2, ..., m$. **A** computes $y_j = (v_j)_{\sigma^{-1}} \oplus s$ for all j.

3. **A** computes the signature: $yxor = y_1 \oplus y_2 \oplus \ldots \oplus y_m$

4. **A** sends *yxor* and σ to **B**.

5. **B** inverts v_j to obtain $y_j \oplus s$, for $j = 1, 2, ..., m$

6. **B** computes $Hy_j = H(y_j) = (H(y_j \oplus s)) \oplus i$

7. **B** verifies that $H(yxor) = (Hy_1 (\text{mod } 2)) \oplus (Hy_2 (\text{mod } 2)) \oplus \ldots \oplus (Hy_m (\text{mod } 2))$

Comments:

- This scheme takes advantage of the distributive property of H(*yxor*).

$H(y_1 \oplus y_2 \oplus \ldots \oplus y_m) = Hy_1 \oplus Hy_2 \oplus \ldots \oplus Hy_m$

- The commitments are message dependent.

Security of the Signature Scheme:

- The security of the signature scheme is due to the difficulty in finding:

$yxor \ni H(yxor) = z$, where H and z are known.

- NOT SECURE $yxor = y_1 \oplus y_2 \oplus \ldots \oplus y_m = (y_1 \oplus s) \oplus (y_2 \oplus s) \oplus \ldots \oplus (y_m \oplus s) \oplus s$

S can be deduced from yxor and the other information

Comments:

- B demonstrates a knowledge of *yxor* and hence *s*.
- Still has not addressed the need to demonstrate that wt(s)=p.
- Question: How big does *m* have to be to guarantee security (clearly $m>1$)?

  Maybe $m = 2$ is good enough.

Computational Requirements:

1. Computation of the v_j (both **A** and **B**).
2. Inversion of v_j (**A**).
3. Computation of *yxor* (**A**).
4. Computation of $H(y_j)=(H(y \oplus s)_j) \oplus i$ (**B**).
5. Verification by **B.**

Communication Requirements:

In addition to the message itself:

- *n* bits are transmitted for *yxor* and
- *u* bits (say about 120) are transmitted representing the seed of the generator that produces σ.

DISTRIBUTION:

3	MS 0449	C. L. Beaver, 6514
1	0449	T. J. Draelos, 6514
3	1072	R. A. Gonzales, 1735
3	0449	V. A. Hamilton, 6514
1	1072	K. K. Ma, 1735
1	1072	R. D. Miller, 1735
1	0449	R. C. Schroeppel, 6514
1	0741	S. G. Varnado, 6500
1	0829	E. V. Thomas, 12323
1	0188	LDRD Program Office, 1030 (Attn: Donna Chavez)
2	0899	Technical Library, 9616
1	0612	Review & Approval Desk for DOE/OSTI, 9612
1	9018	Central Technical Files, 8945-1